计算几何 —— 算法与应用

Mark de Berg
Otfried Cheong
Marc van Kreveld
Mark Overmars 著

邓俊辉 译

清华大学出版社

目录

前言

二十世纪七十年代末，计算几何学（computational geometry）从算法设计与分析中孕育而生。今天，它不仅拥有自己的学术刊物和学术会议，而且形成了一个由众多活跃的研究人员组成的学术群体，因此已经成长为一个被广泛认同的学科。该领域作为一个研究学科之所以会取得成功，一方面是由于其涉及的问题及其解答本身所具有的美感，而另一方面，也是由于在（诸如计算机图形学、地理信息系统和机器人学等）众多的应用领域中，几何算法都发挥了重要的作用。

解决许多几何问题的早期算法，要么速度很慢，要么难于理解与实现。随着近年来一些新的算法技术的发展，此前的很多方法都得到了改进与简化。这本教材力图使得这些现代的算法能够为更广泛的读者理解和接受。本书既是面向计算几何课程的一本教材，同时也可用于自学。

本书的结构。除《导言》外，这 16 章中的每一章都从来自应用领域的某一实际问题入手。这个问题将被转化为一个纯粹的几何问题，进而通过计算几何所提供的方法加以解决。每章所讨论的，实质上就是对应的那个几何问题，以及解决该问题所需要的概念与方法。我们根据所希望覆盖的计算几何专题，来选取有关的应用；而就具体的应用领域而言，这些介绍还远远不够全面。引入这些应用的目的，只是为了激发读者的兴趣；而各章本身的目的，并不在于为这些问题提供现成可用的解决

方法。虽然如此，我们还是认为，为有效地解决应用中的几何问题，计算几何方面的知识是非常重要的。希望本书不仅能够吸引来自算法学术圈的那些人，而且对来自应用领域的人们亦是如此。

同一几何问题，可能有好几种不同的解决方法，不过，在论述大多数几何问题时，我们将只给出其中一种。我们通常所选取的，都是最易于理解与实现的方法。我们也十分注意，尽力使本书能够涵盖更多的方法，比如分治策略、平面扫描以及随机算法（randomized algorithm）等等。对每个问题可能的种种变型，我们也不打算面面俱到；我们觉得，更重要的是首先对计算几何中的各个主要问题做一介绍，而不是过于深入地去探究少数专题的细枝末节。

某些章的若干节标有星号。这些节的内容涉及解法的改进与扩展，或者解释了不同问题之间的相互关联。就对后续章节的理解而言，它们并不十分重要。

每章最后，都由名为“注释及评论”的一节进行概括总结。这些节会给出对应各章所介绍结果的来龙去脉，概述其它的解决方法、一般化处理方法及改进，并给出参考文献。虽然这些节可以被跳过，但是对于那些希望就某一章的专题做进一步了解的读者来说，其中的材料都是非常有用的。

每章后面，都附有一定数量的习题。其中一些旨在检查读者对内容的理解程度，也有些是对书中内容的推广，需要精心解答。高难度的问题以及对应于标有星号各节的问题，也被标上星号。

课程大纲。尽管在很大程度上，本书各章之间是相互独立的，但在进行介绍时，最好还是不要随意打乱其次序。例如，第 2 章介绍了平面扫描算法，故在阅读采用了这一方法的其它各章之前，最好首先了解该章的内容。出于同样考虑，在进入有关随机算法的各章之前，也应该首先阅读第 4 章。

如果是作为计算几何的第一门课程，建议（教师）按照书中的次序来讲授前十章。根据我们的经验，这十章覆盖了任何一门计算几何课程都必须介绍的概念和方法。如果还有可能顾及更多的内容，可以在后面六章中进行挑选。

先修要求。做为教材，本书既适用于高年级本科生课程，也适用于低年级研究生课程，具体安排视课程的其它要求而定。读者应具备算法设计与分析、数据结构的基本知识：必须熟知大 O 记号，以及诸如排序、二分查找和平衡查找树等基本的算法技术。读者不需要对这里所涉及的应用领域有所了解，也几乎不需要什么几何知识。在对随机算法进行分析时，会用到一些非常基本的概率理论。

实现。本书中的算法都是以伪代码的形式给出，虽略显概括笼统，但也算详尽，实现起来相对容易。值得一提的是，我们还尝试着介绍了处理退化情况的方法，在具体实现过程中如不能解决好这一问题，往往会使整个计划落空。

我们认为，动手实现其中一个或多个算法将十分有益；这可以令你获得对算法复杂度的实际感受。每一章都可以当成一个编程训练的课程项目。根据可利用时间的多少，你既可以只实现算法本身，也可以连同应用系统一起完成。

为了实现一个几何算法，若干基本的数据类型——点、直线和多边形等——以及对其实施操作

的一些基本例程都是必需的。实现这些基本例程并使之具有鲁棒性，绝非易事，为此需要投入大量的时间。自己动手这样做一次不无裨益，然而如果能够找到一个提供基本数据类型及其操作例程的现成的软件库，将很有帮助。在我们的万维网页面上，可以找到指向这类软件库的链接。

万维网站。本书还附有一个万维网站，该网站提供了本书各个版本的勘误、所有插图、所有算法的伪代码，以及一些其它资源。其地址是：

http://www.cs.uu.nl/geobook/

如果您发现了书中的错误，或是对本书有何建议，可以通过该页面与我们联系。

关于第三版。第三版的改动主要有两处：第 7 章“Voronoi图：邮局问题”中，增加了关于线段Voronoi图、最远点Voronoi图的讨论；第 12 章“空间二分：画家算法”中，针对低密度场景的BSP树，作为实际输入模型的导论，增加了一节。此外，更正了大量瑕疵与错误（请参阅网站提供的第二版勘误）。每章的“注释及评论”一节也做了更新，以体现新的研究成果及相关文献。为不致影响学生继续在课程学习中沿用第二版，第三版尽可能没有改动原先各节与各习题的编号。

致谢。编写教材是一项耗时的工作，即便有四位作者共同合作，也不例外。在过去几年中我们得到了很多人的帮助：关于本书应该包括、不应该包括哪些内容，有些人提供了有益的建议，有些人在阅读初稿后对如何修改提出了建议，另一些人则指出并更正了前两版中的错误。感谢所有这些人，特别要感谢 Pankaj Agarwal、Helmut Alt、Marshall Bern、Jit Bose、Hazel Everett、Gerald Farin、Steve Fortune、Geert-Jan Giezeman、Mordecai Golin、Dan Halperin、Richard Karp、Matthew Katz、Klara Kedem、Nelson Max、Joseph S. B. Mitchell、Rene van Oostrum、Gunter Rote、Henry Shapiro、Sven Skyum、Jack Snoeyink、Gert Vegter、Peter Widmayer、Chee Yap 和 Gunther Ziegler。感谢 Springer-Verlag 出版社给予的建议和支持，使得本书各版本得以出版，并被译成日文、中文及波兰文。

最后，还要感谢荷兰科学研究组织（Netherlands Organization for Scientific Research – N. W. O.）与韩国研究基金（Korea Research Foundation – KRF）的大力支持。

2008 年 1 月

Mark de Berg
Otfried Cheong
Marc van Kreveld
Mark Overmars

1

计算几何：导言

正漫步于校园的你，突然需要打一个紧急电话。在遍布校园的各个公用电话中，你当然想找到离自己最近的那部。然而，哪一部才是最近的呢？一张校园地图将能帮忙，无论你身处何处，都可以在地图上找到最近的公用电话。这张地图可能会将整个校园划分成不同区域，每个区域都对应着一部最近的公用电话（如 图 1-1 所示）。这些区域形状如何？又该如何计算出它们呢？

尽管这算不上一个至关重要的问题，它却简要描述了一个主要的几何概念，而这一概念在众多应用中都扮演着重要的角色。

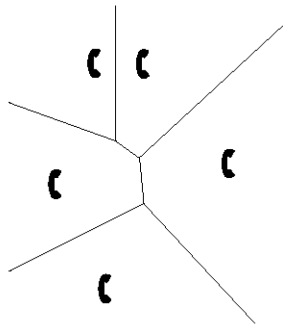


图1-1 按照公用电话的分布，可以将校园划分为若干区域

对校园如此划分之后，就得到了所谓的Voronoi图（Voronoi diagram），第 7 章将详细探讨这一结构。借助该结构，可以为覆盖多个城市的商业区域建立模型，指挥机器人，甚至描述和模拟晶体的生长过程。为了构造Voronoi图之类的几何结构，需要一些几何算法（geometric algorithm）。这些算法就是本书的主题。

第二个例子。假设你已经找到了最近的公用电话。只要手中有一份地图，你很容易就能沿着一条很短的路径到达电话的位置，而且中途不会撞上墙或者其它障碍物（如 图 1-2 所示）。然而，想要通过程序让机器人自己来完成这一任务，却要困难得多。

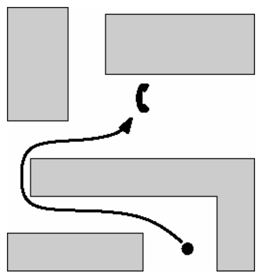


图1-2 从当前位置通往某一公用电话的最短路径

与上例相同，这一问题的实质也是几何的：给定一组几何形状不同的障碍物，我们需要在不与任何障碍物发生碰撞的前提下，找出联接于任意两点之间的最短通路。这就是所谓的运动规划（motion planning），在机器人学中，这类问题的求解至关重要。第 13 和 15 章将针对运动规划所需的几何算法做一讨论。

第三个例子。假设你可以利用不止一张地图，而是两张：一张描述了各个建筑物，包括公用电话；另一张则画出了校园内的道路。为了规划出通往公用电话的运动路径，我们需要将这两张地图叠合（overlay）起来——也就是说，需要将这两张地图所提供的信息合并起来。在地理信息系统（geographic information system）中，地图的叠合是基本的操作之一。这种操作涉及到某张地图中的对象在另一张地图中的定位、不同特征物之间的求交计算等问题。第 2 章将讨论这一问题。

许多几何问题的解决都要依靠精心设计的几何算法，上面只是其中的三个例子。诞生于 20 世纪 70 年代的计算几何（computational geometry），正是旨在解决这类几何问题。这一学科可定义为“针

对处理几何对象的算法及数据结构的系统化研究”，其重点在于“渐进快速的精确算法”。由几何问题带来的挑战吸引了众多的研究人员。从对问题的明确表述，到得出高效而优雅的解决方法，往往需要经历漫长的过程，其间既要克服诸多困难，也要积累一些次优的中间结果。今天，我们已经掌握了功能广泛的一整套几何算法，它们不仅高效而且相对更易理解和实现。

本书将介绍计算几何中最重要的那些概念、方法、算法以及数据结构，但愿我们的介绍方式，能够吸引那些有志于将计算几何的研究成果付诸实际应用的读者。每一章都从某一实际问题入手，而这种问题的求解，都需要借助几何算法。为了说明计算几何之应用范围的广泛性，这些问题分别选自不同的应用领域：机器人学、计算机图形学、CAD/CAM 以及地理信息系统。

然而你并不能指望，在解决应用领域中的主要问题时，总是有现成的软件可以直接利用。这里的每一章，只是孤立地对计算几何中的某一特定概念进行讨论；其中所涉及的应用问题，只是做为一个例子，用以导出有关的概念，继而展开介绍。这些例子可以使我们体会到，如何才能针对工程性问题建立（数学）模型，并进而得出严谨的解答。

1.1 凸包的例子

面对具有几何本质的算法问题，我们所采用的解决方法大多需要具备两方面要素：一是对该问题的几何特性的深刻理解，二是算法和数据结构的合理应用。要是对某个问题的几何性质尚不甚了解，那么纵然有世界上所有的算法在手，你依然不能高效地解决它。反过来，如果不知道有哪些算法技术适用于这个问题，那么即使你已经对问题的几何特性烂熟于胸，也是枉然。通过本书，你将对最为重要的若干几何概念以及算法技术的有个透彻的理解。

为了说明在几何算法的建立过程中所出现的问题，本节将讨论曾在计算几何中首先研究的问题之一——平面凸包的计算。我们在这里忽略该问题的来由；对此有兴趣的读者，可以阅读第 11 章（该章讨论的是三维凸包问题）的导言部分。

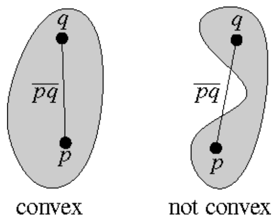


图1-3 凸集与非凸集

平面的一个子集S被称为是“凸”的，当且仅当对于任意两点p, q ∈ S，线段 $\overline{pq}$ 都完全属于S（若 图 1-3 所示）。集合S的凸包$\mathcal{CH}$(S)，就是包含S的最小凸集——更准确地说，它是所有包含S的所有凸集的交。

这里所要讨论的，是如何计算平面上由n个点组成的有限集合P的凸包。可以借助一个虚构式实验，来想象这种凸包的模样：如 图 1-4 所示，将这里的点想象成钉在平面上的钉子；取来一根橡皮绳，将它撑开围住所有的钉子，然后松开手——啪地一声，橡皮绳将紧绷到钉子上，它的总长度也将达到最小。此时，由橡皮绳围住的区域就是P的凸包。

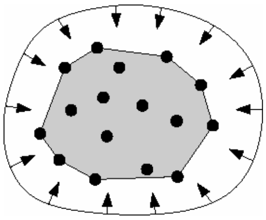


图1-4 凸包的直观理解

因此，也可以将平面有限点集 P 的凸包定义为：顶点取自于 P 且包含 P 中所有点的那个唯一的凸多边形（convex polygon）。当然，这一定义是否有歧义（也就是说，此多边形是否唯一），以及这一定义是否等同于前面所给出的那个定义，都需要严格地予以证明，然而鉴于这是本章的导言，我们将跳过这一环节。

如何来计算凸包呢？回答这一问题之前，必须先回答另一问题：所谓“计算凸包”，到底是什么含义？正如我们已经看到的，P 的凸包是一个凸多边形。表示多边形的一种自然的方法，就是从任一顶点开始，沿顺时针方向依次列出所有顶点。因此，我们所要求解的问题就变成：

给定平面点集 $P = \{ p_1, \ldots, p_n \}$**，通过计算从 P 中选出若干点，它们沿顺时针方向依次对应于** $\mathcal{CH}(P)$**的各个顶点。**

input = **平面上一组点：** $p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

output = **凸包的表示：** p_4, p_5, p_8, p_2, p_9

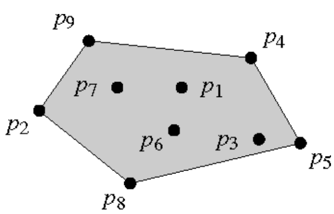


图1-5 计算凸包

当着手设计一个计算凸包的算法时，此前所给出的凸包定义对我们没有多少帮助。按照那个定义，需要计算出“包含 P 的所有凸集的交”，可是这种集合有无限多个。而我们所观察到的“$\mathcal{CH}(P)$是一个凸多边形”这一事实，则更有帮助。下面就来看看，$\mathcal{CH}(P)$是由哪些边构成的。

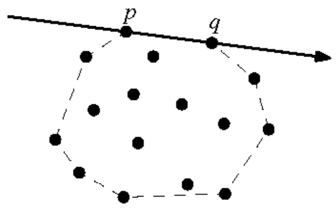


图1-6 相对于$\mathcal{CH}(P)$边界上任一边所在的直线，P中所有点均居于同侧

这些边的端点p和q都来自于P；另外，只要适当地定义由p和q所确定直线的方向，使得$\mathcal{CH}(P)$总是位于其右侧，那么P中的所有点也都将落在该直线的右侧（如 图 1-6 所示）。反之亦然：如果相对于由p和q确定的直线，$P \setminus \{p, q\}$中的所有点都位于右侧，那么$\overline{pq}$就是构成$\mathcal{CH}(P)$的一条边。

好了，在对该问题的几何特性有了更深的理解之后，就可以构造一个算法了。我们通过伪代码来描述该算法，本书将统一采用这种伪代码的形式。

```
算法 SlowConvexHull(P)
输入：平面点集 P
输出：由 CH(P)的顶点沿顺时针方向排成的队列 L
1.    E ← ∅
2.    for (每一有序对(p, q) ∈ P × P，p ≠ q)
3.        do valid ← true
4.            for (除 p 和 q 之外的所有点 r ∈ P)
5.                do if (r 位于 p 和 q 所确定有向直线的左侧)
6.                    then valid ← false
7.            if (valid) then 将有向边pq加入到 E
8.    根据集合 E 中的各边，赵出 CH(P)的所有顶点，并
      按照顺时针方向将它们组织为列表 L
```

或许，你对该算法中的两个步骤还不甚清楚。

第一处出现在第 5 行：如何进行比较，才能判断某个点到底是位于一条有向直线的左侧，还是右侧？对大多数几何算法而言，这都是必需的基本操作之一。本书将假定这些操作都是现成的。显然，（从理论上分析）它们都可以在常数时间内完成，因此从渐进复杂度的角度看，算法的具体实现方法不会对其运行时间的数量级有何影响。但这并不等于说，这些基本操作不甚重要，或者不值一提。实际上，正确实现这些操作并非易事，而且它们对算法的实际运行时间的确会有影响。幸运的是，支持这些基本操作的软件包现已随处可得。因此总而言之，不必去担心如何实现第 5 行中的测试；可以假定我们已经拥有一个子函数，（通过调用该函数）可以在常数时间内完成这类测试。

该算法需要解释的另一个问题，出现在最后一行。通过第 2~7 行的循环，可以构造出凸包的边

集 E。根据 E，可以按照如下方法构造出列表 *L*。E 中各边都是有向的，因此可以定义它们的起点与终点。在指定每条边的方向时，我们都使得其它的所有点都位于它的右侧——这样，如果按照顺时针方向遍历（traverse）所有顶点，那么每条边的起点都会先于其终点被枚举出来。

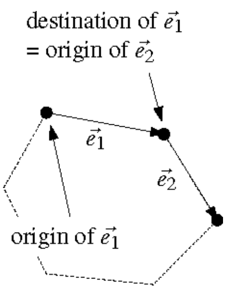


图1-7 确定E中各边的次序

现在如 图 1-7 所示，在E中任意删除一条边$\overrightarrow{e_1}$，将$\overrightarrow{e_1}$的起点、终点分别做为第一、第二个点放入*L*；从E中找出以$\overrightarrow{e_1}$的终点为起点的边$\overrightarrow{e_2}$，将$\overrightarrow{e_2}$从E中删去，并将其终点插入到当前*L*的末尾；再找出以$\overrightarrow{e_2}$的终点为起点的边$\overrightarrow{e_3}$，将$\overrightarrow{e_3}$从E中删去，也将其终点插入到当前*L*的末尾；……。不断重复上述过程，直到E中只剩下最后一条边。至此已经大功告成——因为，最后这条边的终点必然就是$\overrightarrow{e_1}$的起点，而该点已经加入到*L*中了。若直截了当地实现，这一过程需要*O*(n^2)时间。虽然将这一复杂度改进至*O*(n**log**n)并不困难，但是毕竟算法的整体复杂度已经由其它部分决定了。

SLOWCONVEXHULL 算法的复杂度并不难分析。总共要检查 $n^2 - n$ 对点。对每一对点，要检查其它的 n – 2 个点，看看它们是否都位于（该点对所确定有向线段的）右侧。这总共需要运行 *O*(n^3)时间。最后一步需要 *O*(n^2)时间，故总体的时间复杂度为 *O*(n^3)。在实际应用中，这样一个需要运行三次方时间的算法，除非是处理小规模的输入集，否则都会由于太慢而毫无用处。之所以会出现这种问题，是因为我们没有采用任何精巧的算法设计技术，而只是以一种蛮力的（brute-force）方式，将我们对算法的几何理解直接转换为算法。实际上，只要对该算法做进一步的审视，就不难找出改进的方法。

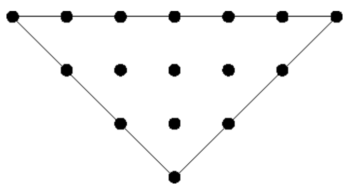


图1-8 多点共线的退化情况

前面介绍了一个准则，籍以判定点对p和q是否定义了*CH*(P)的一条边。然而，在推导这个准则时，我们做得还不够细致。如 图 1-8 所示，相对于由p和q所确定的直线，一个点r的位置并不是非左即右——有时，可能正好落在这条直线上——这就是所谓的“退化情况”（degenerate case），或者简称

为“退化”（degeneracy）。对于这种情况，应该如何处理呢？在刚开始思考某个问题的时候，我们更愿意（暂时地）忽略这些情况，这样，在从问题中抽取出其几何性质的过程中，才不致于把思路搞乱。然而在实践中，这些情况都有可能发生。例如，当借助鼠标在屏幕上标定点的位置时，每个点的坐标都将局限在很窄的一段整数区间之内，因而很有可能会定义出三个共线的点。

在可能出现退化情况时，为了保证算法始终运行正确，就必须这样来重新表述上述准则：某条有向边$\overrightarrow{pq}$是$\mathcal{CH}$(P)的一条边，当且仅当相对于由 p 和 q 所确定的有向直线，所有的其它点 r ∈ P 或者严格地位于其右侧，或者落在开线段$\overline{pq}$上（假定 P 中没有相互重合的点）。这样，此算法第 5 行所涉及的测试，将被替换为一个更为复杂的版本。

还有一个重要的方面也被忽视了，而它却会影响到我们的算法所得出结果的正确性。不知不觉中，我们已经做了这样一个假定：只要给定一条（有向）直线，以及另外一个点，那么无论这个点是位于该直线的左侧还是右侧，我们总是能够准确地做出判断。然而，这个假设并不见得一定成立——如果各点的坐标都表示为浮点数，而且计算过程中所采用的也是浮点运算（floating point arithmetic），那么就必然存在舍入误差（rounding error），从而影响到测试的精度。

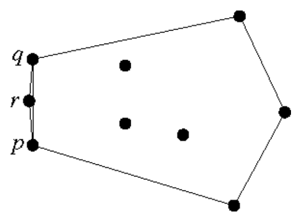


图1-9 三点几乎共线，且与其它诸点相距足够远时，可能选出多余的边

如 图 1-9 所示，试想有三个点p、q和r几乎共线，而其它各点与它们都相距很远。按照上面的算法，要分别对点对(p, q)、(r, q)和(p, r)进行测试。既然这三个点几乎共线，则由于舍入误差的存在，判断的结果很有可能是：r位于直线$\overline{pq}$的右侧，p位于直线$\overline{rq}$的右侧，而q位于直线$\overline{pr}$的右侧。显然，这种几何位置关系是不可能的——然而浮点运算可不管这些！在这种情况下，算法将会把这三条边全都挑选出来。

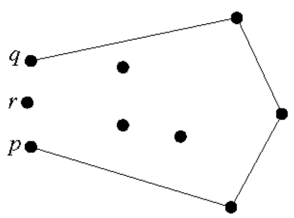


图1-10 三点几乎共线，且与其它诸点相距足够远时，可能会遗漏边

更糟糕的情况是，这三次测试的结果也可能正好与上面相反，这样，如 图 1-10 所示，算法就

会将这三条边都排除掉，于是在所生成的“凸包”边界上将会出现一个缺口。

等到算法的最后一步——将凸包的顶点组织为有序表——时，这将会导致严重的错误。实际上，这一步有个假定：在凸包的每一顶点处，出边和入边都正好只有一条。然而受到舍入误差的影响，某个顶点 p 可能会有两条出边，也可能根本就没有出边。在上面那个简单的算法中，最后一行并没有考虑到对不一致数据的处理，因此，若直接按照该算法编程实现，程序就可能会崩溃。

即使已经证明该算法是正确的，而且也能够处理各种特殊情况，它可能依然称不上鲁棒（robust）——也就是说，计算过程中出现的某个微小误差，可能会导致运行失败，而且失败的形式难以预料。问题的症结在于，在证明算法正确性的时候，我们（想当然地）做了一个假设：可以精确地使用实数进行计算。

我们设计出了自己的第一个几何算法。它可以计算平面点集的凸包。然而，它的时间复杂度为 $O(n^3)$，故运行速度相当慢；它处理退化情况的能力也很差；此外，也不够鲁棒。因此，我们应该尽力去做得更好。

为此，我们将采用一种标准的算法设计模式——递增式策略——来设计一个递增式算法（incremental algorithm）。顾名思义，我们将逐一引入 P 中各点；每增加一个点，都要相应地更新目前的解。这个递增式方法将沿用几何上的习惯，按照由左到右的次序加入各点。于是，首先需要根据 x-坐标对所有点进行排序，产生一个有序的序列：p_1, …, p_n。接下来，我们将按照这一顺序，将它们逐一引入。本来，既然是自左而右地进行处理，所以要是凸包上的顶点也能按照它们在边界上出现的次序自左向右地排列，将会更加方便。然而，情况并没有这样好。因此，我们将首先计算出构成上凸包（upper hull）的那些顶点。

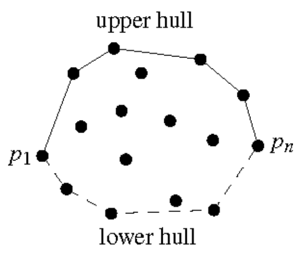


图1-11 分别构造上凸包和下凸包

如 图 1-11 所示，所谓的上凸包，就是从最左端顶点p_1出发，沿着凸包顺时针行进到最右端顶点p_n之间的那段。换而言之，组成上凸包的，就是从上方界定凸包的那些边。此后，再自右向左进行一次扫描，计算出凸包的剩余部分——下凸包（lower hull）。

该递增式算法的基本步骤，就是在每次新引入一个点 p_i 之后，对上凸包做相应的更新。也就是说，已知点 p_1, ⋯, p_{i-1} 所对应的上凸包，计算出 p_1, ⋯, p_i 所对应的上凸包。可以按照如下方法进行。

若按照顺时针方向沿着多边形的边界行进，则在每个顶点处都要改变方向。若是任意的多边形，则每次的转向既可能是向左，也可能向右。然而若是凸多边形，则必然每次都是向右转。根据这一点，在新引入 p_i 之后，可以进行如下处理。令 $\mathcal{L}_{upper}$ 为从左向右存放上凸包各顶点的一个列表。首先，将 p_i 接在 $\mathcal{L}_{upper}$ 的最后——既然在目前已经加入的所有点中，p_i 是最靠右的，则它必然是（当前）上凸包的一个顶点，所以这样做无可厚非。然后，再检查 $\mathcal{L}_{upper}$ 中最末尾的三个点，看看它们是否构成一个右拐（right-turn）。若构成右拐，则大功告成，此时（更新后的）$\mathcal{L}_{upper}$ 记录了组成上凸包的各个顶点 p_1，…，p_i，接下来，就可以继续处理下一个点——p_{i+1}。然而，若最后的三个点构成一个左拐（left-turn），就必须将中间的（即倒数第二个）顶点从上凸包中剔除出去。

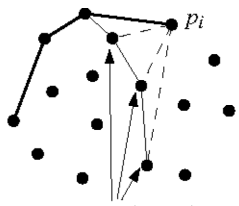


图1-12 只要最后的三点构成左拐，即将居中的点删除

若出现这种情况，需要做的可能还远不止这些——因为，此时的最后三个点可能仍然构成一个左拐（如 图 1-12 所示）。果真如此，就必须再次将中间的顶点剔除掉。这一过程需要反复进行，直到位于最后的三个点构成一个右拐，或者仅剩下两个点。

下面将给出该算法的伪代码。这段代码既计算上凸包，也计算下凸包。在完成后一项工作时，只需将各点自右向左排列，后续的计算与上凸包都是相仿的。

算法 ConvexHull(P)

输入：平面点集 P

输出：由 $\mathcal{CH}$(P)的所有顶点沿顺时针方向组成的一个列表

1. 根据 x-坐标，对所有点进行排序，得到序列 p_1, …, p_n
2. 在 $\mathcal{L}_{upper}$ 中加入 p_1 和 p_2（p_1 在前）
3. **for** (i← 3 **to** n)
4.   **do** 在 $\mathcal{L}_{upper}$ 中加入 p_i
5.   **while** ($\mathcal{L}_{upper}$ 中至少还有三个点，而且最末尾的三个点所构成的不是一个右拐)
6.   **do** 将倒数第二个顶点从 $\mathcal{L}_{upper}$ 中删去
7. 在 $\mathcal{L}_{lower}$ 中加入 p_n 和 p_{n-1}（p_n 在前）
8. **for** (i← n-2 **downto** 1)
9.   **do** 在 $\mathcal{L}_{lower}$ 中加入 p_i
10.   **while** ($\mathcal{L}_{lower}$ 中至少还有三个点，而且最末尾的三个点所构成的不是一个右拐)

11. **do** 将倒数第二个顶点从 L_{lower} 中删去
12. 将第一个和最后一个点从 L_{lower} 中删去
（以免在上凸包与下凸包联接之后，出现重复顶点）
13. 将 L_{lower} 联接到 L_{upper} 后面（将由此得到的列表记为 L）
14. **return**(L)

与上回一样，只要仔细分析，就会发现上面的算法并不正确。不用说，这里同样隐含了这样一个假设：所有点的 x-坐标互异。一旦这个假设不成立，根据 x-坐标所定义的次序就可能有歧义。幸运的是，这一问题实际上并不严重。我们只需以一种合适的方式，对这个次序进行推广——使用字典序，而不是仅仅根据各点的 x-坐标来确定其次序。也就是说，首先按照 x-坐标排序；倘若有多个 点的 x-坐标雷同，则进而按照 y-坐标对它们排序。

还有一种特殊情况被忽略了：如 图 1-13 所示，在对三个点进行比较，以判断它们究竟是构成一个左拐还是右拐的时候，有可能它们恰好共线。

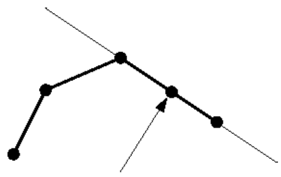


图1-13 三点共线

在这种情况下，居中的那个（那些）点不应该出现在最后的凸包上——因此，应该将共线的点看成是构成一个左拐。也就是说，只有在三个点的确构成一个右拐的时候，我们的测试子程序才应返回“真”；而在其它情况下，都应返回“假”。（请注意，与上面的算法所采用的测试子程序相比，在多点共线的情况下，这种测试要更加容易。）

经过如此修改，我们的算法就能够正确地计算出凸包——如 图 1-14 所示，经过第一趟扫描，构造出上凸包（根据现在的定义，它是从按字典序最小的顶点出发，按照顺时针方向沿着凸包到达字典序最大顶点之间的一段路径）；第二趟扫描则构造出凸包的剩余部分。

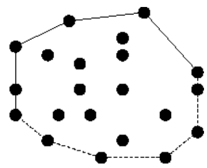


图1-14 由上凸包和下凸包得到凸包

如果由于采用浮点运算而出现舍入误差，这个算法又将如何？若果真发生这类错误，原本应该属于凸包的某个点，就有可能被遗漏掉。不过，该算法输出的结构完整性还不致于受到破坏——也

就是说，它依然能够计算出一个封闭的多边形链。无论如何，算法所输出的顶点列表，总是可以被理解为某个多边形各顶点沿顺时针方向的一个枚举；而且，前后相邻的任何三个点都构成一个右拐（或者，由于存在舍入误差，它们近似地构成一个右拐）。另外，P 中的每个点都不可能与计算出的凸包相距太远。现在，只可能出现一种问题——当某三个点相距很近时，尽管它们构成一个很明显的左拐，却可能会被判断为一个右拐。其后果是，计算出的多边形（的边界）上可能会出现一处凹陷。解决该问题的一种方法，就是要（比如，借助舍入误差）确保相距极近的输入点都能被当成同一个点来处理。这样，尽管最终得到的结果不见得肯定分毫不差，但这种结果毕竟是有意义的——实际上，既然我们采用的本来就是不精确的运算，自然就不可能指望做到百分之百的准确。对许多应用问题而言，这样已经足够好了。当然，在基本测试的实现过程中，细心一些还是明智的，唯此才能尽最大可能地避免错误。

上面的讨论，可以总结为如下定理：

〖定理 1.1〗

给定包含 n 个点的任意一个平面点集，其凸包都可以在 $O(n\log n)$时间内构造出来。

〖证明〗

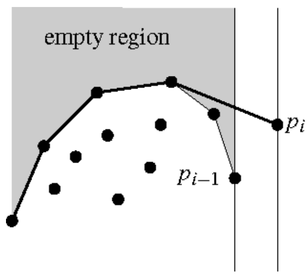


图1-15 在引入p_i后，新链的上方依然是空的

这里只证明对上凸包的计算是正确的；下凸包计算正确性的证明相仿。证明的方法是对处理的点数进行归纳。在**for**-循环开始之前，L_{upper}只包含p_1和p_2两个点，这是平凡的情况，因为$\{p_1, p_2\}$的上凸包就是由这两个点自己确定的。假定L_{upper}中已经存放了$\{p_1, \cdots, p_{i-1}\}$对应的上凸包，现在来考虑加入p_i。在执行完while-循环之后，由归纳假设可知：L_{upper}（中的各点依次）组成一条链，而且该链始终都是右拐。此外，该链起始于$\{p_1, \cdots, p_i\}$中字典序最小的点，终止于字典序最大的点——也就是p_i。为了证明L_{upper}中存放的点就是正确的结果，只需证明：$\{p_1, \cdots, p_i\}$中的各点，要么在L_{upper}中，要么就位于该链的下方。我们可做归纳假设：在引入p_i之前，没有任何点位于此前多边形链的上方。由于此前那条链必定位于新链的下方，故倘若有某个点位于新链的上方，它只可能出现在由p_{i-1}和p_i界定的垂直条形区域中（如 图 1-15 所示）。然而，这是不可能的——因为，果真如此，该点的字典序必然介于p_{i-1}与p_i之间。（你需要按照类似的思路自行验证一下，在p_{i-1}、p_i或者其它点的x-坐标相同时，这个结论依然成立。）

为了证明其时间复杂度的上界（upper bound），请首先注意到，按照字典序对各点进行排序，需要 O(n**log**n)时间。接下来，考虑上凸包的计算。**for**-循环要执行的趟数是线性的。这样，只需要考虑其中 **while**-循环的执行趟数。在每一趟 **for**-循环中，**while**-循环至少要执行一趟。而如果还要额外地执行 **while**-循环，则每趟都会将某个点从凸包中剔除出去。在构造上凸包的整个过程中，每个点至多只能被删除一次，因此，在所有 **for**-循环中（**while**-循环）额外的执行趟数加起来不会超过 n。可以类似地证明，下凸包的计算也至多消耗 O(n)时间。因此，整个计算凸包算法的时间复杂度取决于排序那一步，即 O(n**log**n)。 ❑

最终的凸包算法不仅易于描述，而且也易于实现。它只用到按照字典序的排序，以及对前后相邻三个点的测试（以判断它们是否构成一个右拐）。如果仅仅从该问题最初的表述来看，很难想象得到居然会存在一个如此简单而且高效的解决方法。

1.2 退化及鲁棒性

正如在前一节中已经看到的，一个几何算法的建立过程，往往要经过三个阶段。

在第一个阶段，需要理解我们正在处理的几何概念。然而，我们的思路总是会被一些问题打乱，因此要尽力去忽略这些问题。这类令人讨厌的问题，有的来自多点共线，有的来自垂直线段。在设计或理解算法的最初阶段，暂且将这些退化情况搁到一边，是一种十分有益的策略。

接下来的第二个阶段，必须对前一阶段所设计的算法进行调整，使之即使对退化情况也依然能正确处理。在完成这一任务时，初学者往往会将一大堆的特殊情况引入到算法之中。然而在很多情况下，还有更好的办法——通过对问题的几何性质做再次的分析，往往可以将各种特例集成到一般情况当中。例如在凸包算法中，只要使用字典序来代替 x-坐标顺序，就可以处理多个点具有相同 x-坐标的问题。在本书里的大多数算法中，我们始终都尽量去采用这种集成式的方法，来处理特殊情况。当然，在你初次阅读这些算法的时候，还是不要过于拘泥于这些特殊情况，这样才可以使你更好地理解算法。只有在理解了算法对一般情况的处理过程之后，才能开始考虑有关退化的问题。

只要研读过计算几何方面的文献，你就会发现，许多作者都忽略了特殊情况，为此，他们通常都要对算法的输入做某些特定的假设。还是以凸包问题为例，本来，只要交代一句“假定输入数据中的任何三点都不共线，任何两点的 x-坐标都不相同”，就可以将所有的特殊情况都“排除”掉。从理论的角度来看，这类假设通常都无可厚非——因为，此时的目标，只是要确定某个问题的计算复杂度；而且，即使你能够不厌其烦地去对细枝末节进行讨论，（最终却会发现）各种退化情况几乎无一例外地都能够在不提高算法渐进复杂度的前提下得到处理。然而在具体实现算法时，特殊情况毫无疑问地会增加实际的复杂度。当今计算几何界的大多数研究人员都已经意识到：自己所做的

“一般性位置假设”（general position assumption），在实际应用中并不成立；一般而言，集成式的处理方法是处理特殊情况的最佳方法。此外，还有若干一般性的方法——所谓的“符号扰动法”（symbolic perturbation scheme）。在算法的设计与实现过程中，借助于这类手段，你可以不必考虑退化情况——即使退化情况出现了，算法依然可以正确运行。

最后一个阶段是具体的实现。这时，需要考虑到基本的操作（比如，测试某个点究竟是位于一条有向直线的左侧、右侧，还是落在其上）。要是幸运的话，你可以找到一个现成的几何软件库，其中提供了你所需的那些操作；不然，你只好自己去实现了。

在具体实现的阶段还会出现另一个问题——企图“对实数进行精确运算”是不现实的——因此对于其后果，我们不可不有所了解。在几何算法的实现过程中所遇到的种种麻烦，究其根源，往往可以归结为鲁棒性（robustness）的问题。这类问题非常棘手。有一种方法就是借助于某个（根据具体的问题可能采用整数、有理数甚至代数数来）支持精确运算（exact arithmetic）的软件包，然而运行速度会因此变得很慢。另一种方法是对算法本身作适当调整，使之能够检测到可能出现的不一致问题，并采取适当的措施以避免程序崩溃。然而如此一来，就不能保证算法的输出一定正确，因此，确定其输出的精确性就变得很重要。（这也是前一节设计凸包算法时所做的一项工作——尽管算法给出的多边形有可能并不凸，但我们还是可以肯定，输出的多边形在结构上是正确的，而且它与凸包十分接近。）最后，（后一方法）还可以根据具体的输入，以数值的形式预测出，为了得到问题的正确结果，究竟需要达到多高的精度。

哪一种方法才是最好的，因具体的应用而异。若计算速度不成问题，则精确运算（软件包）更为合适。而在其它一些情况下，算法是否精确不甚重要。比如，若只需显示某个点集的凸包，则即使（绘出的）多边形离真正的凸包稍有偏差，也往往不会被察觉出来。此时，只要在实现时能够做到细心，仍然可以沿用浮点运算的方式。

本书后续章节的注意力将放在几何算法的设计阶段，而不会就其具体实现的阶段再费笔墨。

1.3 应用领域

如前所述，针对每一几何概念、算法及数据结构，本书都挑选了一个能启发读者的应用实例。它们大多源自计算机图形学、机器人学、地理信息系统以及 CAD/CAM 等领域。考虑到部分读者对这些领域还不甚熟悉，这里做一简要介绍，并列举出从这些领域中引发出来的若干几何问题。

1.3.1 计算机图形学

计算机图形学所涉及的问题，是根据建模后的场景生成图像，以输出到计算机屏幕、打印机或

者其它的输出设备。这里的场景，可以简单到二维平面上（由线条、多边形以及其它基本对象组成）的图画（drawing），也可能复杂到（包括光源、纹理之类在内的）具有真实感效果的三维场景。后一类场景极为复杂，其中，多边形或曲面片的数量动辄超过一百万。

在计算机图形学中，既然场景由几何对象构成，几何算法的作用自然就很重要。

二维图形学的问题，通常会涉及到特定（几何）元素的交、确定被鼠标拾取的（几何）元素或者找出位于特定区域之内的所有（几何）元素。在第 6、10 和 16 章中将要介绍的若干技术，对解决其中的某些问题很有用处。

在三维领域，几何问题将变得更加复杂。在显示三维场景时，一个关键的步骤就是隐藏面的消除——找出场景中相对于某个视点的可见部分，或者换而言之，将被各物体遮挡住的部分剔除掉。第 12 章将介绍解决这一问题的一种方法。

为了生成具有真实感的场景，还必须考虑到光照。由此会引发许多新问题，比如阴影的计算。于是，真实感图像的合成就对（诸如光线跟踪、辐射度等）复杂的显示技术提出了要求。而在虚拟现实等需要处理运动物体的应用中，物体之间的碰撞检测（collision detection）也很重要。所有这些实际应用都涉及到几何问题。

1.3.2 机器人学

机器人学研究的是机器人的设计与使用。既然机器人是在三维空间（也就是真实的世界）中工作的物体，其中很多地方就自然会出现几何问题。本章开头已经介绍过运动规划问题：在包含障碍物的某个环境中，机器人需要找出一条路径。在第 13 和 15 章中，我们将讨论运动规划的几种简单情况。运动规划只是任务规划（task planning）的一个方面，后一问题更具一般性。机器人可能会接受到高层次的任务——“清扫房间”——它必须规划出完成任务的最佳方案。这涉及到对运动的规划、对各子任务执行次序的规划，等等。

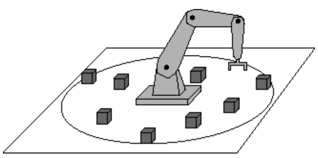


图1-16 工业机器人

在对机器人及其操作的工作零件进行设计时，还会出现其它的几何问题。大多数的工业机器人，不过是如 图 1-16 所示的一只底座固定的机械手。机器手能够对零件进行操作，而给机械手提供零件的方式却很有讲究——应该尽可能使机械手更容易抓住它们。

也许其中的一些零件必须固定，以便机械手对其操作。有些零件则需要转到某个已知的方向，机械手才能对其操作。所有这些都是几何问题，有时甚至还涉及到运动学（kinematics）。本书所介

绍的一些算法，就适用于这类问题。比如第 4.7 节将要讨论的最小包围圆问题（smallest enclosing disc problem），就可以在机械手的最优放置方面派上用场。

1.3.3 地理信息系统

一个地理信息系统（或简称 GIS）存储的是多种地理数据：国土边界、山脉高度、河道走向、植被分布、人口密度或者降雨量，诸如此类。也可能存储一些人工的（地理）结构，比如城市、公路、铁路、电力线路或者煤气管线。借助于 GIS，可以抽取出与某一特定区域相关的信息，尤其是能够获得反映不同类型数据之间关系的信息。例如，有的生物学家可能希望在平均降雨量与某种植物的出现之间建立起某种联系；而在对某个地方进行挖掘之前，土木工程师则可能需要利用 GIS 来确定，那里的地下是否有煤气管道通过。

大多数地理信息系统都涉及到地球表面的点以及区域，因此，几何问题在这个领域屡见不鲜。此外，这里的数据规模非常之大，以至于不得不采用高效的算法。以下就列举出本书将要讨论到的一些 GIS 问题。

第一个问题是：如何存储地理数据？假设拟建立一套汽车导航系统，以便司机能够随时了解自己所处的位置。为此，首先需要将庞大的道路图和其它信息存储下来。任何时候，都要能够在地图上确定汽车所处的位置，并且能够快速地选取图中某个很小的局部，并通过车载计算机显示出来。这些操作都需要高效的数据结构。第 6、10 以及 16 章将分别介绍计算几何解决这些问题的方法。

如 图 1-17 所示，某些山脉地形的高度信息，通常只能在某些离散的采样点测出。而其它位置的高度，则只能通过对邻近的采样点进行（重采样）插值来得到。那么，究竟应该选用哪些点（来进行插值）呢？第 9 章将讨论这个问题。

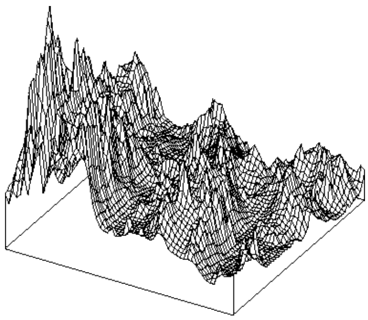


图1-17 山脉地形

在GIS中，不同类型数据之间的组合也是一种至关重要的操作。例如，可能需要检查某个森林中有哪些房屋建筑；也可能需要确定道路与河流的交叉处，以找出所有的桥梁；为了给筹建中的高尔夫球场选择一个好位置，需要找到一块稍有缓坡、价格实惠而且距城镇不远的区域。通常，GIS 都会将不同类型的数据分别存放在不同的地图中。为了将这些数据组合起来，就需要对不同的地图进行叠合（overlay）运算。第 2 章将针对在进行叠合计算时遇到的一个问题进行讨论。

最后，我们将再次引用本章开头所举的那个例子：找出最近的公用电话亭或医院之类的公共设施。为此，需要构造出Voronoi图。第 7 章将详细研究这一结构。

1.3.4 CAD/CAM

计算机辅助设计（computer aided design - CAD）所研究的问题，是如何借助计算机进行产品设计。产品的范围之广，从印刷电路板、机器零件和家具，到完整的建筑物。无论是何种情况，得到的产品都是一个几何实体，因此在这一过程中出现各种各样的几何问题也就不足为怪了。实际上，任何一个 CAD 软件包，都应该能够对物体进行求交与求并运算，能够对物体或物体的边界进行分解，得到形状更简单的子块，还要能够对设计完成的产品做可视化显示。

为了验证某一设计方案是否与需求规范相符，必须进行某些测试。人们往往并不需要为这种测试而建造一个（真实的试验）原型，实际上，只要（在计算机中）进行模拟就足够了。比如第 14 章将讨论在对印刷电路板散热过程的模拟中所出现的一个问题。

某个物体一旦被设计好并经过测试，接下来就可以进行实际的制作。在这一阶段，计算机辅助制造（Computer Aided Manufacturing, CAM）软件包可以大显身手，祝你一臂之力。CAM也会涉及到许多几何问题。第 4 章将讨论其中的一个。

近期的一个研究方向是所谓的“装配设计”（design for assembly），也就是说，在设计阶段就需要考虑将来的装配方案。在支持这一功能的 CAD 系统的帮助下，设计师可以对设计方案的可行性进行验证，以回答与此类似的一些问题：按照某套制造工序，该产品是否能够很容易制造出来？要解决此类问题，离开几何算法将很难想象。

1.3.5 其它应用领域

其它的各应用领域同样都会提出几何问题，而为了解决这些问题，也必须求助于几何算法及其数据结构。

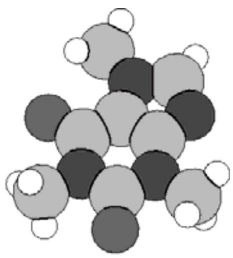


图1-18 咖啡因分子

例如在分子建模领域，通常都是用球体来表示原子，这样，分子就是一堆在（三维）空间中相互联接的球体（如 图 1-18 所示）。此方面经典的问题，将涉及到对所有对应于各原子的球体进行求

并运算，进而得出整个分子的表面（模型），或者计算出两个分子可能相互碰撞的位置。

另一个领域为模式识别（pattern recognition）。例如光学字符识别（optical character recognition - OCR）系统，要求能够在对一张印有文本的稿纸扫描之后，识别出构成文本的字符。在此过程中，一个重要的基本步骤就是将某个字符的图像与一组事先存储好了的字符进行比较，以从中找出最为匹配的那个。这就相应地提出了一个几何问题：给定两个几何对象，如何判断它们的相似程度。

乍看起来，虽然某些领域似乎与几何风马牛不相及，然而它们同样可以由几何算法受益——因为在许多时候，非几何的问题往往都能借助几何的概念而形式化地转换为几何问题。例如，我们将在第 5 章看到：数据库中的每个记录，都可以被理解为高维空间中的一个点；而且，我们还会给出一种基于几何的数据结构——借助于这种结构，对记录的一些查询将会非常高效。

通过上面所介绍的一系列几何问题实例，但愿你能够认识到，在计算机科学众多领域之中，计算几何都扮演了一个重要的角色。本书所介绍的算法、数据结构以及相关技术，将成为你的有力工具，令你在解决几何问题时游刃有余。

1.4 注释及评论

本书中每一章的末尾，都有一节名为“注释及评论”。这些内容将告诉你，在对应章节中所介绍的结果出自何处，有哪些一般性的推广以及改进，并且给出相应的文献索引。当然，也可以跳过这些内容；然而，要是你希望就某一章的主题获得更为深入和广泛的了解，这些节所提供的材料将会对你很有帮助。为了获得更多的相关信息，还可以求助于Handbook of Computational Geometry[331]，或者Handbook of Discrete and Computational Geometry[191]。

本章详细剖析了平面点集凸包的构造问题。这也是计算几何的一个经典问题，有关文献之多可谓汗牛充栋。本章所介绍的算法，通常称作Graham扫描（Graham scan），该算法的最初版本出自Graham[192]之手，这里介绍的是经Andrew[17]修改后的版本。实际上，有很多$O(n\log n)$的算法都可以解决这一问题，这只是其中的一种。Preparata和Hong[322]给出过这样的一个分治算法。还有一个递增式算法 [321]，可以将各点逐一引入，而且每次插入只需$O(\log n)$时间。Overmars和van Leeuwen[305]对这一方法做了推广，无论是插入还是删除顶点，每次操作都只需$O(\log^2 n)$时间。关于这类动态凸包（dynamic convex hull）的结果还有很多，比如Hershberger和Suri[211]的成果。

众所周知，这一问题存在一个$\Omega(n\log n)$的下界（lower bound）[393]，尽管如此，还是有很多人试图突破这一界限。他们的工作并非徒劳无益——因为，在许多应用问题中，实际出现在凸包上的点相对很少；而在建立上述下界的时候，假设了（几乎）所有的点都落在凸包上。因此，寻找一个

运行时间取决于凸包本身复杂度的算法，很是值得。Jarvis[221]提出了一种包扎（wrapping）技术，可以在O(h×n)时间内构造出凸包（其中h为所生成凸包的复杂度），他的这一算法常被称为Jarvis行进（Jarvis march）。基于Bykat[79]、Eddy[156]以及Green和Silverman[193]的工作，Overmars和van Leeuwen[303]也提出了一个算法，其最坏情况的复杂度（与Jarvis行进）一样。不过，这个算法具有一个优点——对于多种随机分布的点集，其期望时间复杂度是线性的。Kirkpatrick和Seidel[238]最终将这一结果改进至$O(n\log h)$；最近，Chan[82]又提出了一个复杂度相同的算法，而且这个算法的实现更为简单。

凸包可以定义在任何维度的空间之中。正如我们将在第 11 章中看到的，三维空间中的凸包也可以在$O(n\log n)$时间内构造出来。然而，在高于三维的空间中，凸包（问题）的复杂度将不再线性正比[①]于输入点的数目。更为详细的介绍，请参见第 11 章的注释及评论部分。

在过去数年之中，提出了许多处理特殊情况的一般性方法。这类所谓的符号扰动法，通过对输入数据做（微小的）扰动，以消除掉其中的退化情况。然而，这种扰动只是在符号上的扰动。这一技术是由Edelsbrunner和Mucke[164]首先提出的；后来，Yap[397]、Emiris和Canny[172][171]又分别做过改进。采用符号扰动法，虽然程序员们可以从处理退化情况的繁重负担中解脱出来，但是这种方法亦非尽善尽美——使用符号扰动法所提供的程序库，算法的速度将会下降；有的时候，还需要从“经过扰动后的结果”中再恢复出“真正的结果”，而这并不总是一桩易事。鉴于这些不足，Burnikel 等人 [78]曾得出结论：还是直接去处理输入数据中的退化情况更好，这不仅（就编程的工作量而言）更加简单，而且（就程序的运行时间而言）也更加高效。

至于几何算法的鲁棒性，是近来才引起人们兴趣的一个问题。大多数的几何比较，都可以形式化地归结为计算行列式的符号。在这种符号计算过程中，为了克服浮点运算的不精确性，有一种方法就是先设定一个很小的阈值ε，如果浮点计算的输出（的绝对值）小于ε，则认为行列式为零。当然，如果直截了当地这样去实现，有可能会导致算法的不稳定（例如，对某三个点a、b和c，可能会判断出$a = b$且$b = c$，但$a \neq c$），并导致程序的运行失败。Guibas等人 [198]指出，只要将这种方法与区间运算（interval arithmetic）和后向误差分析（backward error analysis）结合起来，就可以得出鲁棒的算法。另一种方法是采用精确运算（exact arithmetic）。按照这种方法，在计算行列式符号的时候，需要精确到多少比特位，就真正计算到这样的精度。这同样会令计算速度降低，但好在已经有了多种技术，可以将（为换取精度而）在性能方面的牺牲降至相对很小的程度 [182][395]。除了这些通用的方法外，还有若干篇论文 [34][37][81][145][180][181][219][279]讨论过如何在解决特定问题时实现计算的鲁棒性。

[①] 对任何$\varepsilon > 0$，都有 $n\log n = O(n^{1+\varepsilon})$。从这个意义上讲，$O(n\log n)$可以等同于线性。——译者

本章对一些应用领域做了简要概述，我们正是从这些应用中抽取出一些具体的实例，并由此引出本书所讨论的各种几何概念以及算法。如果想要对各个应用领域做更为深入的了解，你可以参考后面列出的一些参考书。当然，针对这些领域的好书还有很多，这里的介绍只不过是管中窥豹。

有关计算机图形学的书籍非常之多。其中，Foley等人的那本专著 [179]内容详尽而全面，被广泛认为是这方面最好的专著。Shirley等人的 [359]和Watt的 [381]也是很好的参考书。

Choset等人的 [127]，以及Latombe的 [243]和Hopcroft、Schwartz和Sharir的 [217]等略显陈旧的专著中，都对机器人学（robotics）和运动规划（motion planning）问题做过详实的概述。从几何角度关于机器人学的更多信息，可以参考Selig的专著 [348]。

有关地理信息系统的书籍虽然也非常多，但大部分都没有很详细地考虑算法问题。其中有一些通用的教材，比如Demers的 [140]、Longley等人的 [257]以及Worboys与Duckham合著的 [392]。Samet的专著 [335]，则介绍了许多与GIS有关的数据结构。

Faux和Pratt合著的 [175]、Mortenson的 [285]以及Hoffmann的 [216]，在CAD/CAM和几何造型方面都堪称上好的导论性读物。

1.5 习题

习题 1.1 集合 S 的凸包，可以定义为“包含 S 的所有凸集的交”。另一方面，就点集的凸包而言，有人指出：该凸包是（包含这个点集的）周长最短的凸集。我们希望能够证明，这两种定义是等价的。

a. 试证明：两个凸集的交还是凸集——这意味着，有限个凸集的交依然是凸的。

b. 试证明：包含某个点集、周长最短的那个多边形 $\mathcal{P}$，必然是凸的。

c. 试证明：包含点集 P 的任何凸集，都包含上述周长最短的多边形 $\mathcal{P}$。

习题 1.2 给定平面点集 P。令 $\mathcal{P}$ 为包含 P 中所有的点、所有顶点均来自 P 的一个凸多边形。试证明：据此定义，多边形 $\mathcal{P}$ 是唯一确定的；而且，它就是所有包含 P 的凸集的交。

习题 1.3 将某凸多边形各边所对应的 n 条（未排序的）线段组成一个集合 E。试给出一个算法，在 $O(n\log n)$时间内，根据 E 计算出该多边形所有顶点按顺时针方向的一个序列。

习题 1.4 在凸包算法中，必须能够通过测试判断出，相对于由 p 和 q 两点确定的一条有向直线，某点 r 究竟是位于其左侧还是右侧。令 $p = (p_x, p_y)$，$q = (q_x, q_y)$，$r = (r_x, r_y)$。

a. 试证明：通过行列式 $D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$ 的符号，可以判断 r 是位于直线的左侧还是右侧。

b. 试证明：实际上，|D|就是由 p、q 和 r 确定的那个三角形的面积的两倍。

c. 在实现算法 CONVEXHULL 中的基本测试时，为什么上述方法很有吸引力？分别就点坐标为整数或浮点数两种情况，谈谈你的理解。

习题 1.5 通过验证说明：在经过本章所介绍的修改之后，即使是对退化的点集，算法 CONVEXHULL 也照样能够正确地构造出凸包。例如，可以考虑如下令人讨厌的例子：点集中的所有点都共（一条垂直）线。

习题 1.6 在许多情况下，我们需要计算的并不是点集的凸包，而是一组物体的凸包。

a. 给定由平面上的n条线段构成的一个集合S。试证明：S中所有线段的共 2n个端点的凸包，就是S的凸包。

b.* 给定非凸的多边形$\mathcal{P}$[①]。试给出一个算法，在$O(n)$时间内构造出$\mathcal{P}$的凸包。提示：将算法CONVEXHULL做某种变形——比如，不是按照字典序来处理各个顶点，而是按照其它的某种顺序。

习题 1.7 请考虑另一种计算平面凸包的方法：如 图 1-19 所示，从最右端的点开始处理。将该点做为凸包边界上的第一个点p_1。现在，假想有一条通过p_1的直线，从最初的垂直方向开始绕p_1顺时针旋转，直到它碰上另一个点p_2。这个点就是凸包边界上的第二个点。我们继续旋转这条直线（不同的是这次是绕着p_2），直到碰上下一个点p_3，…。

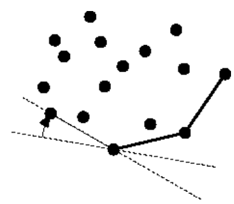


图1-19 礼品包扎

就这样，我们不断旋转这条直线，直到它又重新碰到 p_1。

a. 写出这个算法的伪代码。

b. 会遇到那些退化情况？如何对付它们？

c. 试证明：这个算法能够正确地构造出凸包。

d. 试证明：若实现得当，该算法只需 $O(n\times h)$时间，其中 h 是凸包的复杂度。

e. 若采用的是不精确的浮点运算，可能会出现哪些问题？

习题 1.8 本章所介绍的在 $O(n\log n)$时间内构造平面上 n 个点的凸包的算法，是基于一种递增式的构造模式：逐一加入各点，每加入一点，都要相应地对凸包进行更新。本题的要求是：基于另一种模式——分治模式——来设计来一个算法。

a. 给定互不相交的两个凸多边形 $\mathcal{P}_1$ 和 $\mathcal{P}_2$，其顶点总数为 n。试给出一个算法，在$O(n)$时间内，构造出凸包 $\mathcal{P}_1\cup\mathcal{P}_2$。

① 这里的 $\mathcal{P}$ 应该是所谓简单多边形，即不相邻边不相交的多边形。——译者

b. 以上面你所给出的算法为基础，设计一个分治算法，在 $O(n\log n)$时间内，构造出平面上 n 个点的凸包。

习题 1.9 假定已经有一个现成的子程序 CONVEXHULL，可以为你构造出平面点集的凸包。其输出为凸包上各点按顺时针方向构成的一个序列。现在，给定由 n 个（实）数组成的一个集合 $S = \{x_1, x_2, \cdots, x_n\}$。试证明：可以在 $O(n)$外加调用一次 CONVEXHULL 的时间之内，对 S 进行排序。鉴于排序问题的下界为$\Omega(n\log n)$，故而$\Omega(n\log n)$也是凸包问题的下界。也就是说，就渐进复杂度而言，本章所介绍的算法已经是最优的了。

习题 1.10 在平面上给定由 n 个（可能相交的）单位圆组成的集合 S。我们想要构造出 S 的凸包。

a. 试证明：S 的凸包的边界可以分解为一些直线段和 S 中某些圆上的圆弧。

b. 试证明：S 中的每个单位圆，最多为凸包边界贡献一段圆弧。

c. 令 S'为 S 中各单位圆的圆心所组成的集合。试证明：S 中的某个圆为凸包贡献一段圆弧，当且仅当 S'的凸包边界经过这个圆的圆心。

d. 试给出一个算法，在 $O(n\log n)$时间内构造出 S 的凸包。

e.* 试给出一个算法，即使 S 中各圆的半径不同，也能够在 $O(n\log n)$时间内构造出 S 的凸包。

2

线段求交：专题图叠合

对于身处异国他乡的游客而言，再没有什么要比地图更具价值的信息来源了。地图可以告诉你，游客们对哪些地方最感兴趣；它们也会告诉你，要沿着哪些公路与铁路，才能到达这些名胜景点；它们还能指示出小湖泊的位置，等等。不幸的是，有时地图也会令你失望，因为经常会很难找到你所需的信息：纵然你知道某个小镇的大致方位，也可能很难在地图上确定它的具体位置。为了增加地图的可读性，地理信息系统将（不同类型的）信息划分为若干层（layer）。每一层都是一幅专题图（thematic map）——也就是说，存放某一特定类型的信息。这样，某一层可能负责存储有关公路的信息，第二层存放的可能是所有城市的信息，而另一层则存放河流的信息，诸如此类。某些层对

应的主题（theme）有可能非常抽象。例如，可能会有某一层对应于人口密度的分布、平均降雨量、大灰雄的栖息地（如 图 2-1 所示）或者植被的分布。



图2-1　大灰雄栖息地的分布

各层所记录的地理信息，在数据类型上可能差别极大：对应于道路图的那层，可能会将道路存储为一组线段（或者，也可能是曲线）；对应于城市的那一层，可能由一系列的点组成，每个点分别标有某个城市的名称；而在对应于植被分布的那层中，存放的可能是地图的一个子区域划分（subdividsion），其中的每个子区域分别标有对应的植被类型。

地理信息系统的用户，可能会（从中）选取出若干幅专题图进行显示。比如，为了找到某个小镇，你可能会取出存放城市信息的那一层——这样，诸如河流、湖泊的名称等信息，才不致于分散你的注意力。而在已经确定了该镇的准确位置之后，你可能又需要知道如何达到那里。为此，正如图 2-2 所示的那样，地理信息系统会允许用户察看若干幅地图的叠合（overlay）。



图2-2　加拿大西部的城市、河流、铁道线，以及它们叠合后的效果

借助于道路图与城市图的叠合，你就可以确定前往该镇的路线。在同时显示两幅或多幅专题图层的时候，（不同层在）叠合中相交的位置，往往就是人们最关心的地方。例如，若同时显示对应于道路图的一层以及对应于河流的另一层，则要是能够将所有的相交之处都清晰地标定出来，必将非常有用。在这个例子中，两幅图都基于网络结构，而且它们相交于若干个（离散的）点。而在另外一些场合，人们感兴趣的则是完整子区域之间的交。例如，研究气候的地理学家们的兴趣，可能就会放在寻找那些有松林覆盖、年均降雨量介于 1000 至 1500mm 的子区域。这些子区域，也就是在植被分布图中被标记为“松林”的那些子区域，与在降雨量分布图中被标记为“1000~1500”的那些

子区域之间的交。

2.1 线段求交

接下来，首先要对地图叠合问题的最简单形式做一讨论。也就是说，（相互叠合的）两个地图层，分别都是由一组线段表示的某个网络。以 图 2-3 为例，可能有若干小比例图层分别存放道路、铁路和河流信息。注意，可以通过若干条线段来近似一条曲线。这些线段将导出的一些子区域，不过在此我们对这些子区域并不感兴趣。后面将会考察更为复杂的情况——（相互叠合的）地图不是网络，而是平面经过子区域划分之后导出的、具有明确含义的一些子区域。

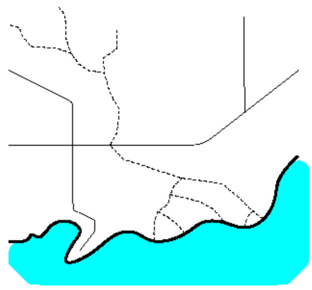


图2-3 道路、铁路及河流图层的叠合

为解决网络叠合的问题，首先需要用几何的概念来描述这一问题。就两个网络的叠合而言，对应的几何条件是这样的：给定由线段组成的两个集合，计算出来自于其中一个集合的所有线段与来自于另一集合的所有线段之间的交点。对此问题的这一定义还不甚明确——什么样的情况，才能称作“两条线段相交”？我们并没有明确定义。比如说，要是一条线段的一个端点落在另一条线段上，是否可以算作相交？换而言之，必须明确地说明，输入的线段究竟是开的，还是闭的。为了确定这一标准，必须回到最初（引出该问题的）应用——网络叠合问题。无论是道路图中的道路，还是河流图中的河流，都可以表示为一条（由依次相联的）线段（组成的）链，因此所谓“道路与河流的交汇点”，就对应于一条链的内部与另一条链的内部的交点。

但这并不等于说，交点总是相对于两条线段的内部而言才出现的——交点也可能碰巧出现在链中的任何一条线段的端点处。事实上，这种情况并不罕见——比如蜿蜒崎岖的河流，就需要用大量的短线段来表示，于是地图在经过数字化处理之后，这些线段的端点的坐标就可能出现舍入误差。由此可以得出结论，应该将这里的线段定义为闭的——这样，要是某条线段的端点正好落在另一条线段上，也会被认为是一个交点。

为了做进一步的简化，我们还将把分别来自两个集合的线段合到一起，构成一个集合，然后来考虑其中所有线段之间的交点。按照这种方式，的确可以计算出所有希望找出的交点。此外，我们还希望找出最初来自同一个集合的各线段之间的交点。当然，这肯定能够办到——因为在我们的应

用中，来自同一集合的线段必然会形成若干条链，沿着每条链，线段依次首尾相联，所以按照我们的定义，不同线段端点的重合，也将被视为交点。在计算完成之后，逐一检查报告出来的每个交点，看看与该交点相关的两条线段是否来自同一个集合——这样，就可以将这部分多余的交点悉数清除出去。于是，我们的问题可以定义为：给定由平面上 n 条闭线段构成的一个集合 S，报告出 S 中各线段之间的所有交点。

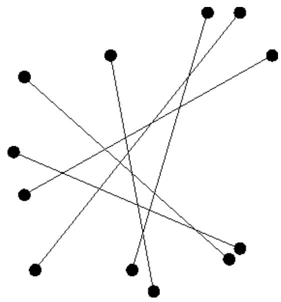


图2-4 最坏情况下，所有线段都两两相交，于是至少需要Ω(n^2)时间

乍看起来，这个问题并没有什么挑战性——可以依次检查每一对线段，看看它们是否相交；如果的确相交，就将其交点报告出来。显然，这种直截了当式的算法需要$O(n^2)$时间。就某种意义而言，这个结果甚至是最优的：若如 图 2-4 所示的那样，每两条线段都相交，那么无论采用什么算法，至少都需要$\Omega(n^2)$时间——因为，哪怕只是直接地逐一报告出所有的交点，也需要这样长的时间。即使是考虑两个网络之间的叠合，也可以构造出这样一个类似的例子。然而在实际的环境中，大多数的线段要么根本不与其它线段相交，要么只与少数的线段相交，因此，交点的总数远远达不到平方量级。要是由某个算法能够在这种情况下计算得更快，那就太好了。也就是说，我们所希望得到的算法，其运行时间不仅取决于输入中线段的数目，还取决于（实际的）交点数目。这样的算法，被称为“输出敏感的”算法（output-sensitive algorithm）——也就是说，这种算法的运行时间对（实际）输出的大小很敏感。也可以称这样一个算法是“交点敏感的”（intersection-sensitive）——因为，输出的大小就是由交点的数目决定的。

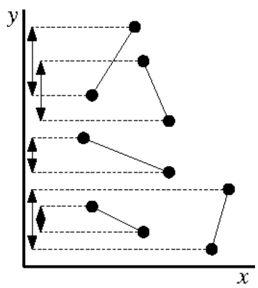


图2-5 通过投影，排除不可能相交的线段对

在找出所有交点的过程中，如何才能避免对所有的线段对进行测试呢？这里必须利用这种情况的几何特性——只有那些相互靠近的线段，才可能会相交；而相距甚远的线段则不可能相交。

下面我们将看到，应该如何利用这一观察结果，得出一个解决线段求交问题（line segment intersection problem）的输出敏感的算法。

设需要对其进行求交的线段构成集合$S := \{s_1, s_2, \ldots, s_n\}$。我们希望避免对相距很远的线段对进行求交。然而具体应该如何做呢？首先排除一种简单的情况。如 图 2-5 所示，将一条线段在y-轴上的正交投影，定义为它的y-区间（y-interval）。任何两条线段，只要其y-区间没有重叠部分——此时，也可以说，它们在y-方向上相距很远——它们就一定不会相交。这样，只需要对那些y-区间相互有所重叠（即与同一条垂线相交）的线段对进行测试。为找出这些线段对，可以想象着用一条直线l，从一个高于所有线段的位置起，自上而下地扫过整个平面。在这条假想的直线扫过平面的过程中，跟踪记录所有与之相交的线段——后面将解释其详细实现——以找出所需的所有线段对。

这类算法被称为平面扫描算法（plane sweep algorithm），其中使用到的直线 l 被称为扫描线（sweep line）。与当前扫描线相交的所有线段构成的集合，被称为扫描线的状态（status）。随着扫描线的向下推进，它的状态不断变化，不过，其变化并不是连续的。

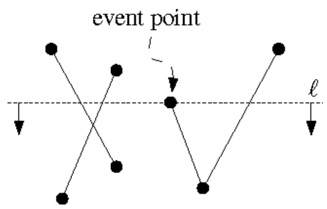


图2-6 平面扫描算法

只有在某些特定的位置，才需要对扫描线的状态进行更新。我们称这些位置为平面扫描算法的事件点（event point）。就本算法而言，这里的事件点就是各线段的端点。

只有在扫描线触及某个事件点的时候，算法才会进行实质的处理——更新扫描线的状态，并进行一些相交测试。具体地，若事件点为某条线段的上端点，则意味着这条线段将开始与扫描线相交，因此需要将该线段插入到状态结构（status structure）中。然后，需要将这条线段，和那些与当前扫描线相交的其它线段分别进行测试，确定是否相交。若事件点为某条线段的下端点，则意味着这条线段将不再与扫描线相交，因此需要将该线段从状态结构中删去。按照这样的方式，只需要对那些可能与某条水平直线同时相交的线段对进行测试。不幸的是，这还不够——因为，在某些（特殊的）情况下，尽管实际的交点数目很少，却依然需要对平方量级的线段对进行测试。一个这样的简单例子就是，所有的线段都是垂直的，而且都与 x-坐标轴相交。因此，目前的这个算法还算不上是输出敏感的。问题在于，与同一扫描线相交的两条线段，在水平方向上仍然有可能相距很远。

在考虑到水平方向的临近性后，我们可以沿着扫描线，将与之相交的所有线段自左向右排序。这样，只有当其中的某两条线段沿水平方向相邻时，才需要对其进行测试。这就意味着，每引入一

条线段，只需要将其与另外的两条线段（具体地讲，就是与新线段上端点左、右紧邻的那两条线段）进行测试。此后，当扫描线向下推进到某个新的位置时，与某条线段紧邻的邻居有可能会发生变化，此时，需要将它与新的邻居进行测试。在算法的状态结构中，这一新策略应该有所反映——现在，状态结构不仅要记录与当前扫描线相交的所有线段，而且还要对这些线段**排序**。为适应新的要求，状态结构不仅要在线段的端点处进行更新，在各交点处，也要做更新——因为在这些位置，（与扫描线）相交的各线段（中有至少两条）的次序必然会有所变化（如 图 2-7 所示）。

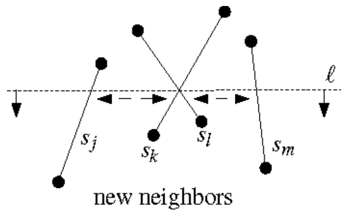


图2-7 每经过一个交点，当前激活的线段之间的次序必然发生变化

在这种情况下，需要找出位置发生变化的那两条线段，然后将它们与各自的新邻居进行测试。这样，就出现了新的一类事件点。

在将这些构思落实为高效的算法之前，我们需要确定，这种方法的确是对的。在需要进行测试的线段对减少之后，是否还能够将所有的交点都找出来呢？换而言之，对于任何两条相交的线段 s_i 和 s_j，是否总存在某个位置，当扫描线 l 抵达该位置时，s_i 和 s_j 沿着 l 是紧邻的？我们还是先忽略掉一些“棘手”的情况——我们假设：没有水平线段；任何两条线段最多相交于一点（也就是说，任何两条线段都不会有局部的相互重叠）；任何三条线段不会相交于同一点。虽然稍后我们就会看到，这些情况都是很容易处理的，但是在目前，还是暂且置之不理的好。至于某条线段的端点落在另一条线段上的情况，等到扫描线触及相应的端点时，这类交点也不难被检测出来。这样，唯一剩下的问题就是：线段之间在内部的每一交点，是否都能被检测出来？

〖引理 2.1〗

设两条非水平的线段 s_i 和 s_j 只相交于其内部的一点 p，而且，任何第三条线段都不经过 p。则在（扫描线到达）高于 p 的某个事件点处（时），s_i 和 s_j 必然会彼此紧邻，并因此接受相交测试（于是对应的交点将被发现）。

〖证明〗

如 图 2-8 所示，令l为比p略高的一条水平线。只要l与p相距足够近，则沿着l，s_i和s_j必然是紧邻的（更准确地说，没有任何事件点落在我们所取的l上，而且也没有任何事件点夹在l与通过p的水平线之间）。

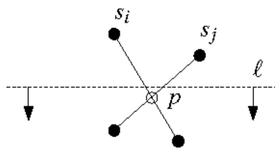


图2-8 交点事件发生前的一刹那

总而言之，必然存在某个位置，当扫描线到达这个位置时，s_i 和 s_j 是紧邻的。另一方面，在算法开始的时刻，s_i 和 s_j 并不是紧邻的——因为在此时，扫描线的位置比所有的线段都高，故状态结构还是空的。因此，必然存在某个事件点 q，在 q 的位置，s_i 和 s_j 开始变为紧邻的，从而接受相交测试。 ❑

这样，就确定了算法是正确的——至少，在暂不考虑此前所提及的那些手的情况时，它是正确的。现在，可以进一步完善我们的平面扫描算法。让我们对整个算法做一扼要重述。假想有一条水平线 1 自上而下扫过整个平面。在某些事件点，扫描线会停留片刻；就目前的算法而言，事件点既包括（事先就可以确定的）各线段端点，也包括（在算法运行过程中逐步发现的）交点。在扫描线移动的过程中，要维护一个有序序列，该序列由所有与当前扫描线相交的线段组成。每遇到一个事件点，扫描线都会停留片刻。此时，上述线段序列会有所变化，因此，必须通过一些动作，对状态结构进行更新，并检测出新的交点——具体的处置方法，取决于事件点的类型。

若事件点对应于某条线段的上端点，就意味着将有一条新的线段开始与扫描线相交（如 图 2-9 所示）。新引入的这条线段必须经过测试，以判断它是否与沿扫描线与之紧邻的另外两条线段相交。

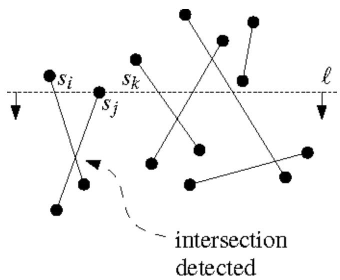


图2-9 上端点事件的处理

只有位于当前扫描线下方的那些交点，才需要加以考虑；至于高于当前扫描线的那些交点，在此之前必然已经被检测出来了。例如，若沿着扫描线，线段 s_i 和 s_k 原本是紧邻的，而（在某个时刻，）第三条线段 s_j 的上端点出现在它们之间，则此时就必须分别将 s_j 与 s_i 和 s_k 进行测试。在检测出来的（最多两个）交点中，只要位于当前扫描线的下方，就是一个新的事件点。在处理完该上端点之后，将继续考虑下一个事件点。

若事件点对应于某个交点，则如 图 2-10 所示，有关的两条相交线段就会交换其（沿扫描线的）次序。它们各自可能（最多）有一条新的紧邻线段，因此，必须分别将它们与其各自的新邻居进行测试，以找出可能的交点。

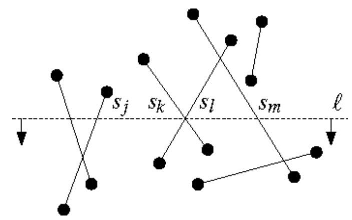


图2-10 交点事件的处理

与上面同理，我们只对位于当前扫描线下方的交点感兴趣。假设在扫描线触及 s_k 和 s_l 的交点那一时刻，有四条线段 s_j、s_k、s_l 和 s_m 依次出现在扫描线上。此后，s_k 和 s_l 将交换次序，于是需要分别对 s_l 和 s_j、s_k 和 s_m 进行测试，以找出它们可能位于扫描线下方的交点。当然，若果真找到了这样的交点，它们也应该属于算法中的事件点。然而值得注意的是，这些事件有可能在此前已经被发现了——比如，有可能某两条线段在此前的一段时间内曾经是紧邻的，后来一度不再紧邻，最终又再次变成是相互紧邻的。

若事件点对应于某条线段的下端点，则它此前的（一左一右）两个邻居现在就会变成是相互紧邻的，因此需要对它们进行相交测试。若它们果真相交，且交点位于当前扫描线的下方，则该交点也将成为一个事件点（同样地，这个事件点也可能在早先已经被发现过）。如 图 2-11 所示，假设（在某一时刻）沿着扫描线，有依次相邻的三条线段s_k、s_l和s_m，扫描线继续前移后遇到了s_l的下端点。于是，s_k和s_m将变成是相互紧邻的，因此我们要对它们进行相交测试。

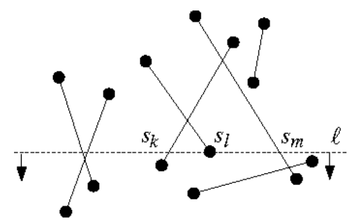


图2-11 下端点事件的处理

在扫描完整个平面之后（更准确地讲，在处理完所有事件点之后），就确定了所有的交点。之所以能够保证这一点，是由于在平面扫描过程中，如下不变性始终成立：（在任何时刻，）处于扫描线上方的所有交点都已经被正确地检测出来了。

在对该算法做了上述简要概述之后，现在需要进行更为详细的讨论。而且，我们也可以顾及到可能出现的各种退化情况（比如，三条或更多条线段交于同一点）。在这些退化情况下，我们期望算法给出什么样的结果呢？必须首先对此做出明确的定义。也许按照我们的要求，算法只要能够逐一报告出各交点，而且每个交点只报告一次就够了；然而要是算法还能够对每个交点，同时给出一个列表，列举出穿过该点（或者以该点为其端点）的所有线段，就将会更有帮助。对于另外一种特殊情况，我们也需要明确地给出定义：在出现这种情况的时候，算法应该给出什么样的输出。这种情况就是，某两条线段的局部有所重叠。不过，为简明起见，本节的后续部分将忽略这一情况。

我们从算法所使用到的数据结构开始介绍。

首先，需要一种数据结构——所谓的事件队列（event queue）——来存放（当前已被检测出来，但尚未发生的）事件。这个事件队列记作Q。我们还需要用到一种操作——把即将发生的下一事件从Q中删除掉，并将它返回（给主程序），以便对它进行处理。这个事件，就是位于扫描线下方、位置最高的那个事件。倘若有两个事件点的y-坐标相同，则约定返回x-坐标更小的事件点。也就是说，位于同一条水平线上的事件点，将按照从左到右的次序接受处理。按照这一约定，若是一条水平线段，则应该将其左（右）端点视为上（下）端点。这一约定也可以这样来理解：我们使用的扫描线不是水平的，而是（沿逆时针方向）略向上方倾斜的（如 图 2-12 所示）。

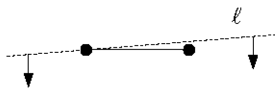


图2-12 左端点优先策略的几何解释

于是，对于水平线段，扫描线将首先触及它的左端点，然后再触及右端点；而且，（只要扫描线的倾角足够小，就能）保证在这两个位置之间，不会触及任何其它的事件点。该事件队列必须支持插入操作——因为在算法的运行过程中，可能会出现新的事件。请注意，不同事件点的位置可能重合。例如，两条不同的线段，其上端点可能重合。显然，将它们做为同一事件点来处理，会更加自然一些。因此，在进行插入操作的时候，必须检查待插入的事件是否已经出现在 Q 中了。

我们这样来实现事件队列。首先，要在各事件点之间定义一个次序$\prec$，各事件点将按照这个次序接受处理。对于任何两个事件点 p 和 q，定义“$p \prec q$ 当且仅当 $p_y > q_y$，或者 $p_y = q_y$ 且 $p_x < q_x$”。所有的事件点将按照由$\prec$确定的次序，组织为一棵平衡二分查找树（balanced binary search tree）。对于 Q 中的每一个事件点 p，我们还同时记录下起始于 p 的（也就是以 p 为其上端点的）那条线段。在对事件进行处理的时候，这方面的信息是必需的。这两种操作——取出下一事件和插入新的事件——每次各需要 $O(\log m)$时间，其中的 m 为 Q 中事件的数目。（这里并没有使用堆来实现事件队列，因为还需要测试某个给定的事件是否已经存在于 Q 之中。）

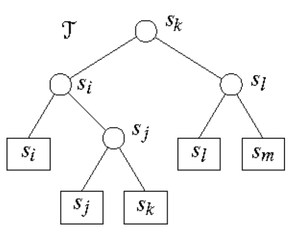


图2-13 用平衡二分查找树来实现状态结构

其次，还需要维护算法的状态。所谓状态，也就是与当前扫描线相交的所有线段构成的有序序

列。借助一个状态结构（status structure），可以访问某一给定线段s的（左、右）邻居——这样，在插入s之后，就可以立即进行相应的相交测试。这个状态结构记作T。它必须是动态的——一旦有某条线段开始（或不再）与扫描线相交，就应将它插入到状态结构中（或从状态结构中删去）。在任一时刻，状态结构中的所有线段之间具有一个定义明确的次序，因此可以使用一棵平衡二分查找树来实现状态结构（图 2-13）。

有些读者也许只用二分查找树存储过数字，因此对于这里对二分查找树的用法，他们难免会有些费解。然而实际上，二分查找树完全可以用来存储任意类型的一组元素——只要在这些元素之间可以定义一个明确的次序。

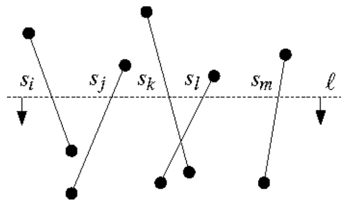


图2-14 任一时刻，与扫描线相交的各线段之间存在明确的左右次序

解释得更详细一点，与当前扫描线相交的每条线段，都按照其次序，存放在该平衡二分查找树T的某匹叶子处。如 图 2-14 所示，（各线段）沿着扫描线自左向右的次序，与T中各叶子自左向右的次序完全一致。在T的各内部节点处，也要存储某些信息，以在进行查找过程中提供必要的指导，最终找到所需的叶子。在每个内部节点处，要存放其左子树中的最右端叶子。（实践中有另一做法：只将各线段存放在这些内部节点处。——这的确可以节省空间。不过，从概念上看，将存放在内部节点处的线段想象成用以引导查找的数值，而不是真正的数据项，将使算法更加易于理解。将线段存放在叶子处，也可以使算法的描述更加简明。）假设某个点p正落在扫描线上，我们需要查找紧邻于其左侧的那条线段。在每个内部节点v处，为了判断p究竟是位于该线段的左侧还是右侧，只要将p与记录在v处的线段做一次比较。根据比较的结果，就可以相应地深入到v的左子树或右子树，直到最终到达某匹叶子。我们所要查找的那条线段，不是存放在这匹叶子处，就是存放在紧邻于其左侧的那匹叶子处。类似地，也可以找到紧邻于p右侧（或者包含p）的那条线段。这样，无论是一次更新，还是对紧邻线段的一次查找，都只需要$O(\log n)$时间。

事件队列 Q 以及状态结构 T，就是我们所需的所有数据结构。至此，整个算法可描述如下：

算法 FINDINTERSECTIONS(S)
输入：平面线段集 S
输出：S 中各线段之间的所有交点（以及穿过各交点的线段的信息）
1. 初始化一个空的事件队列 Q 然后，将所有线段的（上、下）端点插入 Q 中；对于上端点，还要记录其对应的线段

2.	初始化一个空的状态结构 $\mathcal{T}$
3.	**while** ($\mathcal{Q}$ 非空)
4.	**do** 找出 $\mathcal{Q}$ 中的下一事件点 p，将其删除
5.	HANDLEEVENTPOINT(p)

前面已经介绍了不同事件的处理方法：若是线段的端点，则需要在状态结构$\mathcal{T}$中插入或删除线段；若是交点，则需要交换（对应的）两条线段的次序。无论何种情况，在事件发生后，对每一对新近成为邻居的线段，都要进行相交测试。在退化情况下——即某个事件点涉及到多条线段时——具体的实现将更为微妙。下面的子程序，将描述正确处理各类事件点的方法（如 图 2-15 所示）。

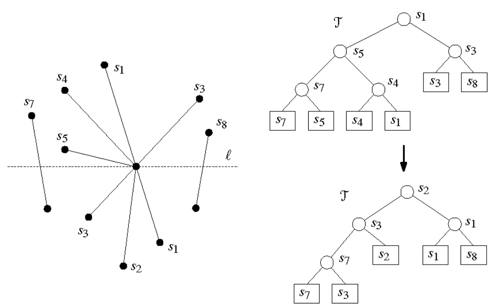


图2-15 在处理一个事件点时，状态结构的相应变化

算法	HANDLEEVENTPOINT(p)
1.	令 U(p)为所有以 p 为上端点的线段构成的集合； 这些线段都与事件点 p 存放在一起 （若是水平线段，则以其左端点做为上端点）
2.	在$\mathcal{T}$中找出包含p的所有线段 [①] (* 在$\mathcal{T}$中，这些线段是（依次）相邻的 *) 在所找出的线段中 将那些以p为下端点的线段组成集合L(p) 将那些在内部包含p的线段组成集合C(p)
3.	**if** (L(p)∪U(p)∪C(p)包含不止一条线段)
4.	**then** 报告“发现交点 p”；同时返回 L(p)、U(p)和 C(p)
5.	将 L(p)∪C(p)中的线段从 $\mathcal{T}$ 中删除
6.	将U(p)∪C(p)中的线段插入到$\mathcal{T}$中 (* $\mathcal{T}$中各线段的次序，必须与它们和扫描线刚离开p之后的相交次序一致 *) (* 若存在水平的线段，则将它排在包含p的所有线段 [②]的最后 *)

[①] 准确地讲，应该是“将 p 包含于内部，或者以 p 为其下端点的线段”。——译者

[②] 准确地讲，应该是“将 p 包含于内部，或者以 p 为其上端点的线段”。——译者

```
7.    (* 将 C(p)中的线段删除，然后按照逆序重新插入 *)
8.    if (U(p)∪C(P) = ∅)
9.        then 在 T 中，找出 p 的左右邻居 s_l 和 s_r
10.            FindNewEvent(s_l, s_r, p)
11.       else 在 T 中，找出 U(p)∪C(p)里最左边的线段 s'
12.            在 T 中，找出与 s'紧邻于左侧的线段 s_l
13.            FindNewEvent(s_l, s', p)
14.            在 T 中，找出 U(p)∪C(p)里最右边的线段 s''
15.            在 T 中，找出与 s''紧邻于右侧的线段 s_r
16.            FindNewEvent(s'', s_r, p)
```

请注意，在第 8~16 行中假定了 s_l 与 s_r 的确存在。否则，相关的那些步骤显然就不必执行。

查找新交点的几个子程序非常简单：它们只要对两条线段进行比较，即可判定它们是否相交。唯一需要小心的是：一旦找到一个交点，则该交点只有两种可能——要么早先就已经处理过了，要么还没有。若没有水平线段，则只要交点位于当前扫描线的下方，它就还没有被处理过。然而，对于水平的线段，又该如何处理呢？请记住这里的约定：对 y-坐标相同的事件，我们将按照从左到右的次序进行处理。这就意味着，对于处于当前事件点（水平）右方的交点，我们依然会感兴趣。因此，FindNewEvent 子程序应该如下定义：

```
算法 FINDNEWEVENT(s_l, s_r, p)
1.    if (s_l 和 s_r 相交于当前扫描线的下方
          （或者交点正好落在当前扫描线上并且在当前事件点的右侧），
          而且该交点尚未做为一个事件出现在 Q 中)
2.        then 将这个交点做为一个事件，插入到 Q 中
```

该算法的正确性如何？显然，FINDINTERSECTIONS 所报告的交点的确是“货真价实的”，然而，是否所有的交点都会被该算法报告出来呢？下面这则引理指出，实际情况的确如此。

〖引理 2.2〗

算法 FINDINTERSECTIONS 能够正确地计算出所有的交点，并能同时给出穿过各交点的线段。

〖证明〗

按照此前的约定，事件的优先级决定于其 y-坐标；若有多个事件的 y-坐标相同，则 x-坐标越小者优先级越高。我们将通过对各事件点的优先级作归纳，来证明该引理。

任取一个交点 p，假定所有优先级更高的交点 q 已经被正确地计算出来了。令集合 U(p)由所有以 p 为上端点（对于水平线段，则是左端点）的线段组成；令集合 L(p)由所有以 p 为下端点（对于水平线段，则是右端点）的线段组成；令集合 C(p)由所有在其内部包含 p 的线段组成。

首先，假设 p 是某条或某几条线段公共的端点。这种情况下，在算法的第一步，p 就已经出现在事件队列 *Q* 中了。来自 U(p)的线段都与 p 存放在一起，因此它们都可以被查找出来。在 p 接受处理的时候，L(p)和 C(p)中的线段已经存储在 *T* 中，因此在 HANDLEEVENTPOINT 的第 2 行，这些线段也会被查找出来。总之，只要 p 是一条或多条线段的共同端点，则 p 以及与之相关的线段都可以被正确地查找出来。

接下来，假设p不是某条线段的端点。只需证明：或早或晚，p必将被插入到*Q*中。需要注意的是，相关的所有线段，都将p包含于内部。将这些线段，按照其围绕p的角度排序，任取其中相邻的两条线段s_i和s_j。由〖引理 2.1〗的证明过程可知，必然存在优先级高于p的某个事件点q，使得在（扫描线）越过q之后，s_i和s_j开始变成是相互紧邻的。，为了简明起见，我们在〖引理 2.1〗中曾经假定：s_i和s_j都不是水平的。尽管如此，那个证明的方法仍然可以直接应用于水平线段的情形。根据归纳假设，事件点q已经被正确地处理过了——也就是说，p已经被检测出来，并存入到*Q*之中。 ❑

现在我们已经知道，这个算法是正确的。不过，我们所设计的这个算法称得上是输出敏感的吗？答案是肯定的——该算法的运行时间为 *O*((n+k)**log**n)，其中 k 为实际输出的规模。接下来的这则引理给出了一个更强的结论：运行的时间为 *O*((n+I)**log**n)，其中 I 为交点的数目。之所以说这个结果更强，是因为对每个交点而言，可能需要输出大量的线段——比如，在有很多线段交汇于同一点时。

〖引理 2.3〗

对于由平面上任意 n 条线段组成的集合 S，算法 FINDINTERSECTIONS 的运行时间都是 *O*(nlogn + Ilogn)，其中 I 为 S 中各线段之间的交点总数。

〖证明〗

算法的第一步，就是将所有线段的端点组成（初始的）事件队列。既然我们使用了平衡二分查找树来实现事件队列，故这一步需要的时间为*O*(n**log**n)。对状态结构的初始化只需要常数时间。接下来，开始进行平面扫描，逐一处理各事件。为了处理某一事件，需要对事件队列*Q*进行（最多）三次操作：在FINDINTERSECTIONS的第 4 行，将该事件从*Q*中删除掉；还需要对FindNewEvent调用一到两次——相应地，最多会有两个新事件加入到*Q*中。*Q*的每次删除或插入操作需要*O*(**log**n)时间。对状态结构*T*，也要进行一些操作——插入、删除以及查找紧邻线段。每一次这类操作也只需*O*(**log**n)时间；而操作的总数线性正比于 m(p) := card

(L(p))∪U(p)∪C(p))——也就是与该事件相关的线段总数。若令m为所有事件点p对应的m(p)之和，则该算法的运行时间就是O(m**log**n)。

显然，m = O(n+k)，其中 k 为输出的规模。无论如何，只要 m(p) > 1，事件点 p 所涉及到的所有线段都会被报告出来，而任一线段所涉及到的事件点，无非就是线段的端点。

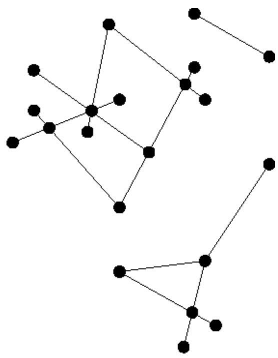


图2-16 输入线段对应的平面图

然而，我们的目标是要证明m = O(n+I)，其中I是交点的总数。为证明这一点，我们将所有输入线段的集合看成是嵌入于平面空间的一幅平面图（planar graph，如果你对平面图这一术语不甚熟悉，可以首先阅读第 2.2 节的第一段）。如 图 2-16 所示，该图的顶点就是各线段的端点以及各线段之间的交点，而其中的边则是联接于各顶点之间的线段（或线段的局部）。现考察某个事件点p。它必然是图中的一个顶点，因而m(p)不会超过该顶点的度数。这样一来，m就不会超过图中所有顶点的度数总和。其中，每条边为两个（而且正好两个）顶点（亦即其端点）分别贡献一度，因此，m不会超过 $2n_e$，其中n_e为图中边的总数。现在，要根据n和I来给出n_e的上界（upper bound）。考虑其中顶点的总数n_v，根据定义，$n_v \le 2n+I$。另外，正如总所周知的，对于平面图而言，必有$n_e = O(n_v)$——这样，该引理的结论就已经得证。不过，为完整起见，在此我们还是给出（有关平面图顶点数与边数关系的）具体论证过程。在平面图中，每张面至少由 3 条边围成（当然，假定至少有三条线段），而每一条边至多与两张面相关联。因此，图中面的总数不会超过$\frac{2n_e}{3}$。现在要借助欧拉公式（Euler's formula）——这个公式指出，对任何由n_v个顶点、n_e条边组成的平面图，若其中的面数为n_f，则必有如下关系成立：

$$n_v - n_e + n_f \ \ge\ 2$$

等号成立的充要条件为“图是连通的”。将前面得出的 n_v 和 n_f 的上界加进来，就得到了

$$2 \ \le\ (2n+I) - n_e + \frac{2n_e}{3} \ =\ (2n+I) - \frac{n_e}{3}$$

于是，$n_e \ \le\ 6n + 3I - 6$，因而 $m \ \le\ 12n + 6I - 12$——于是，本引理关于运行时间的结论得证。 ❑

还需要分析复杂度的另一方面——该算法所占用的存储空间量。每条线段在树 T 中存储至多一份，故该结构只需要 O(n)空间。不过，Q 的规模可能会很大。每发现一个交点之后，都需要将它插入到事件队列 Q 中；每处理完一个交点事件之后，也要删除它。如果要等待很长的时间，交点事件才能处理完毕，那么 Q 的规模就可能会很大。当然，其规模不可能超过 O(n+I)。不过，要是占用空间的大小总是线性的，就更好了。

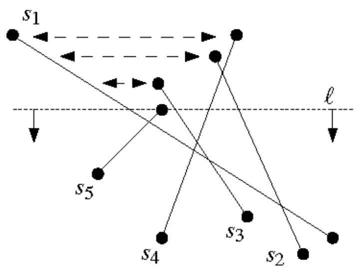


图2-17 随着扫描线的推进，原先相邻的线段可能不再相邻

通过一种简单的方法，就可以实现这样的空间复杂度——任何时刻，只考虑与当前扫描线相交的线段，并只存储其中每一对相邻线段之间可能的交点。在上述算法中，除了这类交点之外，还存储了某些曾经一度水平相邻、但后来不再相邻的线段对之间的交点（图 2-17）。只要始终只存储当前相邻线段之间的交点，Q中事件点的总数就不可能超过线性规模。相应地，算法所需做的改动只有一点——两条（曾经相邻的）线段一旦不再相邻，就立即将它们的交点（从Q中）删去。由于在（扫描线）到达这两条线段的交点之前，它们必然会再次变成是相邻的，所以该交点还是会被报告出来。而且，算法总体的运行时间依然保持为O(n**log**n + I**log**n)。这样，就得到了如下定理：

〖定理 2.4〗

给定由平面上任意 n 条线段构成的一个集合 S。可以在 O(nlogn + Ilogn)时间内，使用 O(n)空间，报告出 S 中各线段之间的所有交点，以及与每个交点相关的所有线段。其中，I 为实际的交点总数。

2.2 双向链接边表

以上已经解决了地图叠合问题的最简单情况——叠合的两幅地图都是各由一组线段表示的网络。一般的地图，结构要更为复杂：实际上要将整个平面划分为多个子区域，各有自己的标记——对整个平面的这样一个子区域划分，才是地图。例如，有关加拿大森林分布的一幅专题图，就是将加拿大划分为若干子区域，分别标记为“松树”、“落叶树”、“桦树”或“混合”等等。

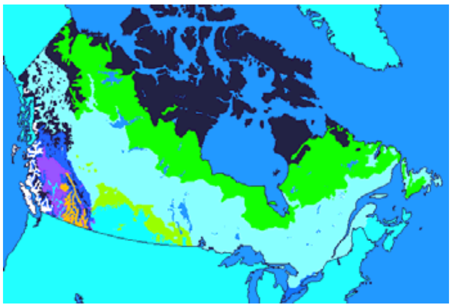


图2-18 加拿大的森林类型分布

为了给出一个算法，以计算出两个子区域划分的叠合，必须首先建立起表示子区域划分的某种适当方式。简单地将一个子区域划分存储为一组线段，并非良策——如此一来，诸如“报告某个子区域的边界”等操作将会十分复杂。最好能够引入结构性的、拓扑的信息，比如：某个给定的子区域是由哪些线段围成的，哪些子区域是相邻的，诸如此类。

我们所考察的，是由图的平面嵌入（planar embeddings of graph）而导出的平面子区域划分（如图 2-19 所示）。只要原来的图是连通的，其对应的子区域划分就必然也是连通的。原图中每个节点（node）的嵌入，称为一个顶点（vertex）；原图中每条弧（arc）的嵌入，称为一条边（edge）。

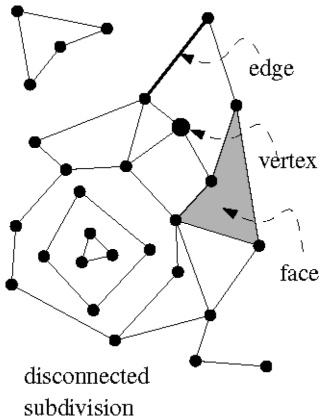


图2-19 由图的平面嵌入而导出的平面子区域划分

我们只考虑一类特殊的嵌入——其中的所有边都必须是直线段。原理上，子区域划分中的边并不见得一定是直的。甚至，有的子区域划分不是任何图的平面嵌入——其中允许出现无界的边。不过本节并不考虑这种一般性的子区域划分。我们将每条边都看成是开的——也就是说，它并不包含自己的（两个）端点（相应地，也就是子区域划分中的顶点）。给定一个子区域划分，其中所谓的“面”（face），指的是在平面上除去所有的顶点和所有的边之后，余下的每一个极大的连通子集。按照这一定义，每张面都是一个开集，其边界由子区域划分的某些边和顶点围成。所谓一个子区域划分的复杂度，就是构成该子区域划分的顶点、边和面的总数。若一个顶点是某条边的端点，就说这个顶点与这条边是关联的（incident）。依此类推，一张面与其边界上的每条边也是关联的，一张面与其边界上的每个顶点也是关联的。

做为子区域划分的一种表示方法，应该满足哪些要求呢？我们可能需要对其实施的一种操作，就是在给定一个点之后，找到该点所在的那张面。在一些应用中，这个操作真是太有用了。实际上，后面的第 6 章将针对这一问题设计一种专门的数据结构。不过，本章所介绍的只是一种基本的表示方法，要想有效地支持这类操作，有点勉为其难。就目前能够获得的信息而言，仍然更多地限制于局部。例如，以下这些操作都是可行的：围绕指定的某张面，沿其边界遍历（traverse）一周；在指定一条公共边之后，通过与其相邻于一侧的面，找到另一侧的那张面。另一种有用的操作是：在给定一个顶点之后，（依次）枚举出与之关联的所有边。下面介绍的表示方法，将能够支持这些操作。这种结构称作双向链接边表（doubly-connected edge list）。

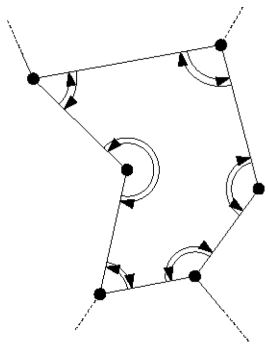


图2-20 通过指针遍历任一张面的边界

对于任一子区域划分，与之对应的双向链接边表为其中的每张面、每条边和每个顶点都设置了一个记录。统而言之，在每个记录中不仅存有几何的、拓扑的信息，而且还有一些附加信息。例如，若某个子区域划分表示的是一张关于植被分布的专题图，则在其对应的双向链接边表中，每张面的记录都将存贮对应子区域的植被类型。附加信息也被称为属性信息（attribute information）。如图 2-20 所示，借助双向链接边表所提供的几何与拓扑信息，即可实施以上的基本操作。为了能够沿逆时针方向围绕某张面遍历一周，可在每条边（对应的记录）中存储一个指针，指向下一条边。当然，有时也需要沿相反方向进行遍历，因此还需要为各边配备另一个指针，指向前一条边。通常，每条边都隶属于两张面的边界，因此还需要为各边配备一对指针（分别指向与之关联的两张面）。

一种便捷的方法是，将每条边的两端分别视为一条半边（half-edge）。这样，任何一条半边都有唯一的一条后继半边、唯一的一条前驱半边。于是，每条半边就只隶属于唯一一张面的边界。如图 2-21 所示，每一条边都被分成两条半边——它们互为孪生兄弟（twin）。在为每一条半边指定后继半边时，总是依照同一原则：后继半边的方向，应该能够沿逆时针方向遍历其对应的面。这样，也同时为各条半边导出了一个方向：如果观察这沿着这一方向前行，每条半边所参与围成的那张面，总是位于其左侧。既然已经定义了半边的方向，我们就可以谈论半边的起点（origin）与终点（destination）。若一条半边$\vec{e}$起始于v，终止于w，则Twin($\vec{e}$)起始于w，终止于v。为找到一张面的边界，还需要在对应的面记录中存放一个指针，指向任一参与围成该面的半边。只要找到了这样一条半边，就可以顺藤摸瓜，通过后继边指针，围绕一张面依次访问边。

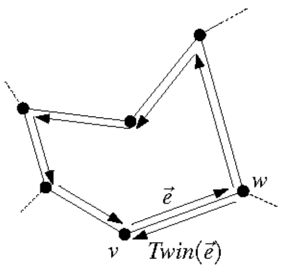


图2-21 孪生半边

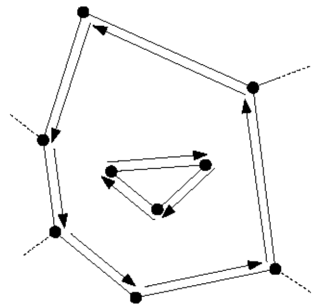


图2-22 沿着空洞的边界逆时针前进，面却总是居于左侧

若某张面中存在空洞（如 图 2-22 所示），则对这种空洞的边界来说，上面所说的性质就不见得成立——比如，在沿逆时针方向遍历这些空洞的过程中，面就总是居于其右侧。当然，如果能够在定义半边方向时，使得与之关联的面总在同一侧，就会更加方便。因此，若是空洞，就按顺时针方向来遍历其边界。这样，无论是哪张面，对于构成其边界的那些半边来说，该面总是位于其左侧。由此可得的另一结论是：任意一对孪生半边，方向必然相反。前面提到过，通过指向其边界上任何一条半边的指针，可以遍历某张面的整个边界；然而要是在某张面中存在空洞，则仅仅通过这一个指针，并不能保证访问到边界上所有的半边。如果某张面的边界由多个连通块组成，就应该为这张面设置多个指针，逐一指向每个连通块。要是某张面中含有孤立（也就是不与任何边相关联）的顶点，也要相应地设置指针，指向这类顶点。为简明起见，对这类情况我们都将不予考虑。

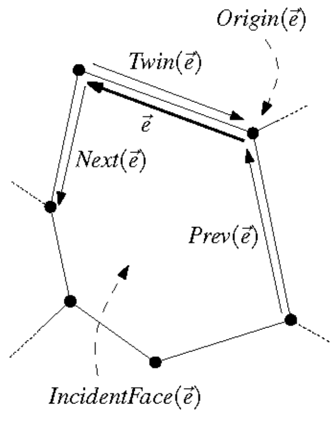


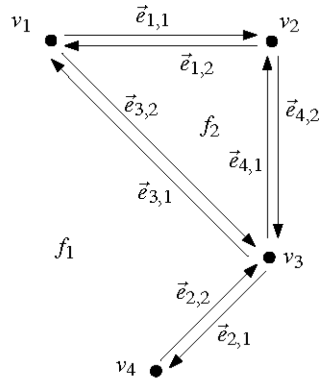
图2-23 DCEL结构的各组成部分

总结一下。如 图 2-23、图 2-24 所示，双向链接边表由三组记录构成：一组对应于顶点，一组

对应于面，还有一组对应于半边。在这些记录中，分别存有下列几何的及拓扑的信息：

- 在对应于顶点 v 的顶点记录中，设有一个名为 Coordinates(v)的域，存放 v 的坐标。此外，还有一个名为 IncidentEdge(v)的指针，指向以 v 为起点的某一条半边。
- 在对应于面 f 的面记录中，设有一个名为 OuterComponent(f)的指针，指向该面外边界（outer boundary）上的任意一条半边。若是无界面（unbounded face），则此指针为 nil。此外，还有一个名为的 InnerComponents(f)的列表，其中设有多个指针，分别对应于该面的各个空洞；每个指针所指的，是其对应空洞的边界上的某一条半边。
- 在对应于半边$\vec{e}$的半边记录中，设有一个名为 Origin($\vec{e}$)的指针，指向该半边的起点；另有一个名为 Twin($\vec{e}$)的指针，指向其孪生半边；还有一个名为 IncidentFace($\vec{e}$)的指针，指向其参与围成的那张面。半边的终点无需存储——因为它等于 Origin(Twin($\vec{e}$))。半边起点的选取，要使得在从该半边起点走向终点的过程中，IncidentFace($\vec{e}$)位于$\vec{e}$的左侧。此外，半边记录中还设有两个指针 Next($\vec{e}$)和 Prev($\vec{e}$)，分别指向其沿着 IncidentFace($\vec{e}$)边界的后继边与前驱边。这样，沿着 IncidentFace($\vec{e}$)的边界，以$\vec{e}$的终点为起点的半边只有 Next($\vec{e}$)一条，而以$\vec{e}$的起点为终点的半边也只有 Prev($\vec{e}$)一条。

每个顶点和每条边所对应的信息量，都是常数规模的。面记录占用的空间可能会更多一些——因为，面 f 中含有多少个空洞，列表 InnerComponents(f)中就需要有多少项。然而，只要将所有的 InnerComponents(f)列表合起来统计，就会发现其中指向任何一条半边的指针都不会超过一个。由此可以得出结论：每个子区域划分所需存储空间的规模，与该子区域划分本身的复杂度呈线性关系。在下面，给出了一个简单的子区域划分所对应的双向链接边表。其中，对应于边 e_i 的两条半边分别记为$\vec{e}_{i,1}$和$\vec{e}_{i,2}$。



Vertex	Coordinates	IncidentEdge
v_1	(0,4)	$\vec{e}_{1,1}$
v_2	(2,4)	$\vec{e}_{4,2}$
v_3	(2,2)	$\vec{e}_{2,1}$
v_4	(1,1)	$\vec{e}_{2,2}$

Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

图2-24 DCEL的数据结构定义

根据双向链接边表所提供的信息，足以完成一些基本的操作。例如，给定一张面，我们可以沿着它的外边界遍历一周——从半边 OuterComponent(f)开始，不断沿着 Next($\vec{e}$)指针，依次访问边界上的各条半边。也可以枚举出与某一顶点 v 相关联的所有边。希望读者能够独立想出实现枚举的具体方法，这是将一次不错的练习。

以上所介绍的双向链接边表，只是该数据结构的一种通用的版本。在某些具体的应用中，顶点本身可能不含任何属性信息，此时，我们就可以将它们各自的坐标，直接存储于与之相关联的边的 Origin()域中，而不必墨守陈规地为顶点记录专门定义一种数据类型。另外，在很多实际应用中，子区域划分中的面并不具有任何含义（只要想一想此前曾经提及的河流网络或者道路网络，就会明白这一点）——认识到这一点更加重要。若果真如此，就可以完全舍弃掉面记录，以及各半边对应的 IncidentFace()域。在下一节我们就将看到，其中介绍的算法并不需要这些域（要是它们不需要更新，实际上还可以实现得更加简单）。在双向链接边表的某些实现中，可能还会要求子区域划分的顶点和边所对应的图必须是连通的。为此，只需引入一些虚边（dummy edge）。保证连通性至少有两个好处：首先，只需一趟图遍历，就可以访问到所有的半边；另外，（既然连通图对应的子区域划分绝不会出现空洞，）InnerComponents()列表也就不必存在了。

2.3 计算子区域划分的叠合

以上设计出了一种能够很好表示子区域划分的方法，现在可以来着手解决一般性的地图叠合问

题。给定两个子区域划分S_1和S_2，其叠合（记作$O(S_1, S_2)$）也是一个子区域划分。$O(S_1, S_2)$有一张面f，当且仅当在S_1、S_2中分别存在面f_1和f_2，使得f是$f_1 \cap f_2$中的一个极大连通子集。这个定义听起来晦涩难懂，可实际上叠合运算本身却极易理解——用白话说就是：所谓叠合，就是由来自S_1和S_2的边共同在平面上导出的一个子区域划分。图 2-25 就是这种运算的一个实例。

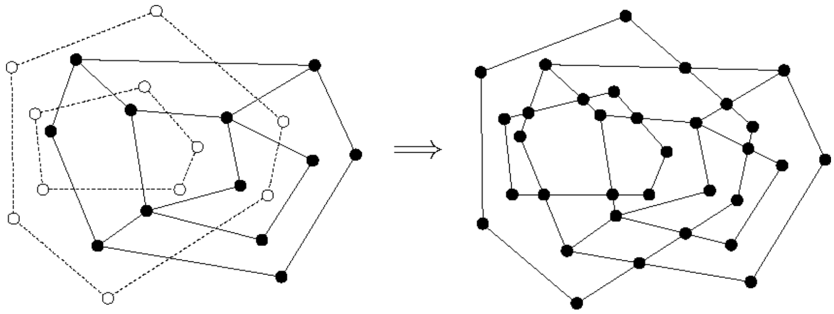


图2-25 两个子区域划分的叠合

所谓一般性的地图叠合问题，就是在给定 S_1 和 S_2 各自所对应的双向链接边表之后，计算出对应于 $O(S_1, S_2)$的双向链接边表。这里还要求为 $O(S_1, S_2)$的每张面都加上标注，以指明在 S_1 和 S_2 中它分别属于哪一张面。如此就能访问到存储在这些面记录中的属性信息。比如，在对植被分布图和雨量图做过叠合之后，就可以知道，在叠合后所得到的各个子区域中，对应的植被类型以及降雨量。

首先来看看，来自 S_1 和 S_2 所对应的双向链接边表的信息，有多少可以被对应于 $O(S_1, S_2)$的双向链接边表继续沿用。考虑由 S_1 的边及顶点构成的网络。这个网络将被 S_2 的边切分为很多块。在这些块中，很大一部分都是可以直接沿用的；只有被来自 S_2 的边分割过的那些边，才需要更新。然而，在对应于这些块的双向链接边表中，各半边记录是否也有这种性质呢？要是某条半边的方向改变了，还必须改变这些记录中的信息。幸运的是，实际情况并非如此。所有半边的方向都定义得很好，以至于它们各自参与围成的面总是处于其左侧。经过叠合，虽然面的形状可能会有所改变，但是它依然处于半边原先的一侧。第一幅地图中的边，有些并没有与第二幅地图中的边相交，根据刚才的分析，这些（未受影响的）边所对应的半边记录，就可以继续沿用。换而言之，在 $O(S_1, S_2)$对应的双向链接边表中，很多半边记录都可以直接从 S_1 和 S_2 中照搬过来；只有与两张地图之间的交点相关联的那些半边，才不能这样直接获得。

根据上述分析，可以得出如下算法。首先，将 S_1 和 S_2 所对应的两个双向链接边表复制到一个新的双向链接边表中。当然，（目前的）这个新的双向链接边表并不是一个合法的双向链接边表——因为，它所表示的还不是某个合理的平面子区域划分。而这正是叠合算法所要完成的任务——通过计算两个边网络之间的交点，并适当地将两个双向链接边表中的相关部分链接起来，最终把这个新的双向链接边表，转化为对应于 $O(S_1, S_2)$的一个合法的双向链接边表。

至此尚未论及新的面记录。这些记录所对应的信息更难计算，因此我们押后讨论这一问题。我

们先来更为详细地介绍一下，如何计算 $O(S_1, S_2)$所对应双向链接边表中的顶点记录和半边记录。

这里采用的算法，将基于第 2.1 节中计算一组线段之间交点的平面扫描算法。如 图 2-26 所示，首先，将来自两个子区域划分S_1和S_2的边合并成为一个集合，然后对这个线段集应用那个算法。在这一步，我们将各边看成是闭的。请记住，这个算法需要两个数据结构的支持——用来存放事件点的事件队列*Q*，以及状态结构*T*。后一结构用一棵平衡二分查找树实现，其中自左向右地记录了与当前扫描线相交的所有线段。

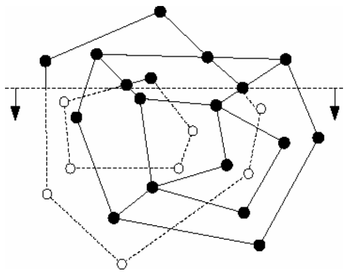


图2-26 基于DECL结构，通过平面扫描解决地图叠合问题

现在，我们也同样维护一个双向链接边表 *D*。刚开始时，*D* 不过是 S_1所对应双向链接边表的一份拷贝，外加 S_2所对应双向链接边表的一份拷贝。在平面扫描的过程中，我们要逐步地将 *D* 转换为对应于 $O(S_1, S_2)$的一个双向链接边表。也就是说，目前暂且只考虑各顶点记录与半边记录，至于面的信息，将在随后计算出来。联接于状态结构 *T* 中各边与 *D* 中各半边记录之间的指针，将被保留下来。这样，在遇到一个交点并因而需要对 *D* 中某些部分做相应改动的时候，我们就能够方便地访问到这些部分。这个过程所具有的不变性就是：在平面扫描的任一时刻，位于扫描线上方叠合的部分，已被正确地计算出来。

现在来考虑一下，在触及一个事件点时，应该如何处理。首先，与线段求交算法一样，要对 *T* 和 *Q* 做更新。若与当前事件点有关的边全部来自同一个子区域划分，则不必再做其它处理——此时，这个事件点就是一个可以继续沿用的顶点。反之，若当前事件点同时涉及到分别来自不同子区域划分的边，则需要通过对 *D* 的局部调整，在这个交点处将原先的两个子区域划分各自对应的双向链接边表链接起来。这个过程虽然十分繁琐枯燥，但难度不大。

我们只介绍其中一种可能情况的详细处理过程。如 图 2-27 所示，这种情况是：S_1的一条边e穿过S_2的一个顶点。此时，应该将边e替换为两条边（分别记为e'和e"）。相应地，在双向链接边表中，e所对应的两条半边就要变成四条。要生成两个都以v为起点的半边记录。如 图 2-27 所示，e原先对应的两条半边依然保留，而且还是分别以e原先的端点为起点。然后，通过设置Twin()指针，将原先的两条半边分别与新生成的两条半边配对。这样，无论是e'还是e"，都各自被表示为一条原先的半边以及一条新生成的半边。

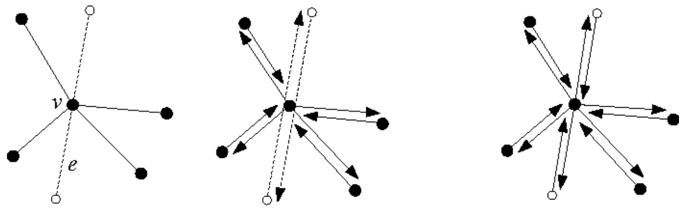


图2-27 来自某个子区域划分的一条边，在另一个子区域划分中穿过一个顶点：在对交点进行处理之前各部分的相对几何位置（左），对应的两个双向链接边表（中），以及在对交点进行处理之后的双向链接边表（右）

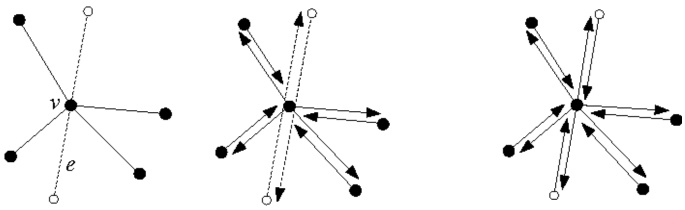


图2-28 设置e两个端点处的链接

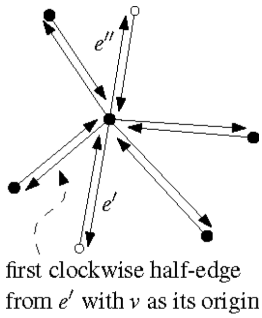


图2-29 设置顶点处的链接

现在，必须进一步设置好若干个Prev()和Next()指针。首先要处理的是e两个端点处的链接设置；稍后，我们再去考虑顶点v处的链接设置。两条新的半边，将分别在原先的半边中找到不是自己孪生兄弟的那条，然后将其Next()指针复制过来。同时还要修改这两个Next()指针所指向的半边，将其Prev()指针指向各自对应的那条新的半边。只要对照 图 2-28，这一步的正确性就一目了然。

接下来，需要对顶点v周围的（链接）设置做更新。在这一局部，某些半边的指针必须重新设置——其中，e'和e''所对应的半边总共有四条，而来自S_2、与v关联的半边也有四条。如 图 2-29 所示，只要通过测试以确定e'和e''在围绕顶点v的环形次序中所处的位置，就可以进一步找到来自S_2 的那四条半边。通过来自一个方向的Next()指针与来自另一个方向的Prev()指针，可以将这四对半边（两两）链接起来。比如，要从e'起沿顺时针方向找到下一条以v为起点的半边，然后将它（通过其Prev()指针）与e'所对应的、以v为终点的那条半边（通过其Next()指针）相互链接起来；要从e'起，沿逆时针方向找到下一条以v为终点的半边，然后将它（通过其Next()指针）与e'所对应的、以v为起点的那条半边（通过其Prev()指针）相互链接起来。e''的处理方法与此相仿。

以上绝大多数步骤只需常数时间，只有一处例外——在围绕 v 的各边中确定 e'和 e''的位置。这一步需要更长时间，其时间复杂度线性正比于 v 的度数。还有其它一些情况，比如，分别来自不同地图的两条边相交，或者顶点重合。不过，这些情况并不比以上讨论的情况更难处理，它们也需要 O(m)时间，其中 m 为与事件点相关的边数。这就意味着，尽管需要再对 $\mathcal{D}$ 进行更新，但整个过程的渐进复杂度仍然没有超过第一步的线段求交算法。请注意，计算出来的每个交点，在叠合之后都是一个顶点。由此可得结论：可以在 O(n**log**n + k**log**n)时间内，计算出 O(S_1, S_2)所对应双向链接边表中的所有顶点记录和半边记录，其中 n 为 S_1 和 S_2 的总体复杂度，而 k 为二者叠合结果的复杂度。

在对涉及顶点和半边的记录设置完毕后，尚待完成的工作就是计算出 O(S_1, S_2)中各张面的信息。更准确地说，针对 O(S_1, S_2)的每张面 f，都需要生成一个面记录；需要将 OuterComponent(f)指针指向其外边界上的某一条边；需要生成一个指针列表 InnerComponents(f)，其中的指针分别指向 f 内部所含的各个空洞（边界上的任一条边）。此外，还要设置好沿 f 边界各条半边的 IncidentFace()域，使它们指向对应于 f 的面记录。最后，在每张新的面中还要做上标记，指明它在原来的两个子区域划分中，分别属于哪一张面。

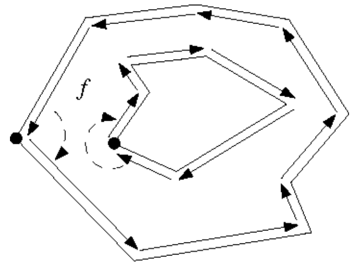


图2-30 外边界与空洞边界的区分

总共有多少张面呢？除无界的那张面外，其余的每张面都有唯一的一圈外边界。因此，需生成的面记录的数目，必然等于所有外边界的数目再加一。根据目前所构造出来的双向链接边表，很容易就可以找出所有的边界环。然而，一个环既可能是（某张面的）外边界，也可能是某张面内部一个空洞的边界。如何区分呢？为此，如 图 2-30 所示，可先在环中找到最左端的顶点（若有多个，则取其中的最低者）。你应该记得，在约定半边的方向时，已保证与之关联的面在（各半边的）局部总是处于左侧。考察在环上与v相关联的那两条半边。既然与之关联的面总是处在左侧，就可以计算出这两条半边朝这张面内部所张的角度。如果这个角度小于 180º，则该环就是一圈外边界；否则，就是某个空洞的边界。在每个环上，只有最左端的顶点才具有这个性质，而其它的顶点则不见得。

既然一张面可能有多条边界，如何判断究竟是哪些环围成了同一张面呢？为此，需要构造一张图$\mathcal{G}$。每一个环，无论是内部的还是外部的，都对应于$\mathcal{G}$中的一个节点。我们假想无界面也有一条外边界，并专门为之设置一个节点，对应于其假想的边界。两条环（各自对应的节点）之间联有一条弧，当且仅当其中一个环为某个空洞的边界，而且另一个环上有一条半边从左侧直接紧邻于空洞边界环上的最左端顶点。若某个环上最左端顶点的左侧根本就没有任何半边，则将该环所对应的节点

联接到无界面所对应的节点。图 2-31 给出了这样的一个例子。

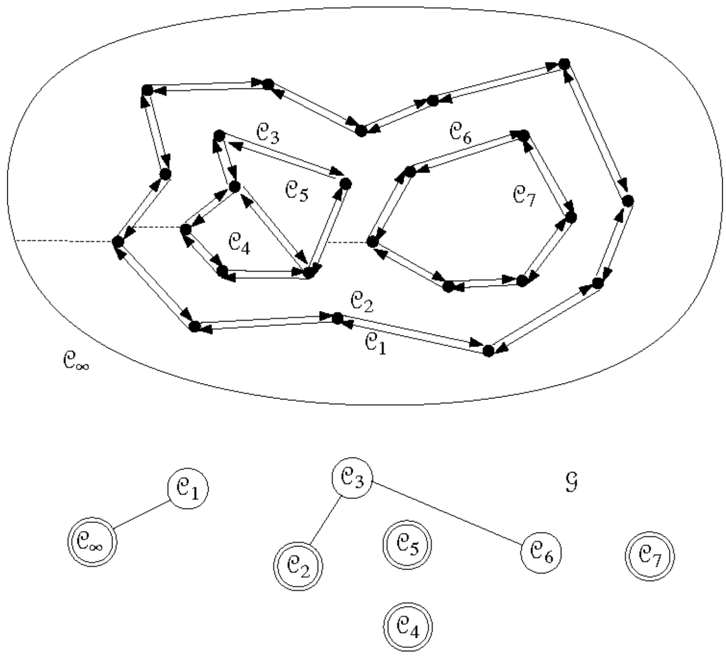


图2-31 子区域划分及其对应的图$\mathcal{G}$

图中的虚线，表示空洞环与其它环之间的联接关系。这张图还给出了与该子区域划分对应的图。空洞环用单线条圆圈表示，而外边界环则用双线条圆圈表示。可看到，C_3 和C_6 都属于C_2 所在的连通子块。这说明C_3 和C_6 是同一张面的空洞环，而且该面的外边界是C_2。若面f只含一个空洞，则图$\mathcal{G}$将该空洞对应的边界环联接到f的外边界。正如可从 图 2-31 看出的，通常情况并非如此——一个空洞也可以联接到另一空洞。位于同一张面f中的这个空洞，既可以联接到f的外边界，也可以联接到另一空洞。然而，正如下面这则引理所指出的，最终还是要把其中的某个空洞联接到外边界。

〖引理 2.5〗

图 $\mathcal{G}$ 中的每一连通子块，都恰好对应于与某张面相关联的所有（内、外）环。

〖证明〗

考虑对应于面 f 中某一空洞边界的一个环 C。既然在此局部 f 位于 C 上最左端顶点的左侧，C 就必然联接到 f 的另一边界环。这样，$\mathcal{G}$ 中同一连通子块中的所有环，都是同一张面的边界。

为最终完成证明，还需反过来说明：f 中每一空洞对应的边界环，都与 f 的外边界环属于同一连通子块。假设有至少一个环不是这样。令 C 为其中（其左端点）最靠左的那个环。由定义，在 C 与部分位于 C 中最左端顶点左侧的另一个环 C'之间，有一条弧相联。于是，C'必与 C 同属一个连通子块，而且该子块不是 f 的外边界所在的连通子块。这与 C 的定义矛盾。 ❑

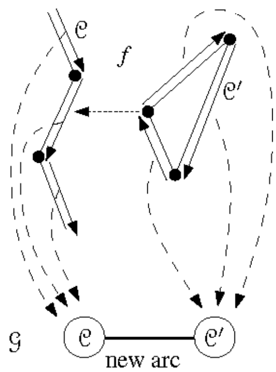


图2-32 找出𝒢的各条弧

根据〖引理 2.5〗，只要有了图𝒢，即可为每一连通子块相应地生成一张面。然后，可为每张面f的各条半边设置好IncidentFace()指针，并构造出InnerComponents(f)列表，以及OuterComponent(f)集合。那么，如何构造𝒢呢？我们记得，在线段求交的平面扫描算法中，总是要找出直接紧邻于各事件点左侧的线段（这种线段要和穿过该事件点的最左侧边进行测试，以确定它们是否相交）。这就是说，为了构造𝒢所需要的信息，早在平面扫描的阶段就已经获得了。因此，为构造𝒢，首先要为每个环指定一个节点。如 图 2-32 所示，为了确定𝒢的各条弧，我们要从每个空洞的边界环上取出最左端的顶点v。若$\vec{e}$为紧邻于v左侧的那条半边，则需找出$\vec{e}$所在的环所对应的节点，以及以v为其最左端顶点的空洞环所对应的节点，然后在它们之间添加一条弧。为了能够在𝒢中高效地找到这些节点，需要为每条半边的记录配置一个指针，指向其所在的环在𝒢中对应的节点。这样，在经过平面扫描之后，只需再花上$O(n+k)$时间，就可以为双向链接边表中的各张面设置好相应的信息。

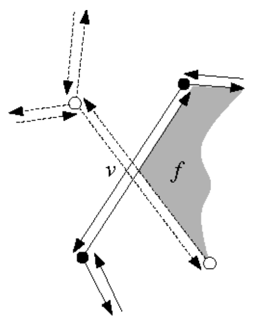


图2-33 找出面f在原先两个子区域划分中所属的面

最后一个问题：在叠合结果中，每一张面f都要做相应的标记，以指明在原先的两个子区域划分中，它分别包含于哪张面中。为了找到这两张面，我们来考虑f的任一顶点v。如 图 2-33 所示，若v是来自S_1的边e_1与来自S_2的边e_2之间的交点，则只要适当地找出e_1和e_2分别对应的半边，然后根据这两条边各自的IncidentFace()指针，就可以确定，在S_1和S_2中f分别包含于哪张面之内。若v不是一个交点，而是原先某个子区域划分（不妨设为S_1）的一个顶点，则我们只知道在S_1中，f被包含在哪张面的内部。为了在S_2中找出包含f的那张面，我们还需要做些工作——必须在S_2中找到包含v的那张面。也就是说，只要知道了S_1中的各顶点分别位于S_2的哪一张面之内，也知道了S_2中的各顶点分

别位于S_1的哪一张面之内，我们就可以正确地给$O(S_1, S_2)$中的每张面做上标记。问题在于，如何计算出这些信息呢？方法是，再一次采用本章所介绍的算法模式——平面扫描。然而，对于最后的这一步，我们将不再做更多的解释。这也是一次很好的练习，通过它可以检查一下，关于采用平面扫描策略来设计算法，你自己的理解有多深。（实际上，根本不需要为此专门进行一次平面扫描，即可计算出这些信息。在此前进行求交计算的那次平面扫描中，这项任务可以一并完成。）

以上分析可归纳为如下算法：

算法 MAPOVERLAY(S_1, S_2)

输入：用双向链接边表形式存储的两个平面子区域划分S_1和S_2

输出：S_1与S_2的叠合，也存储为一个双向链接边表$\mathcal{D}$

1. 生成一个新的双向链接边表$\mathcal{D}$，
   将S_1和S_2对应的两个双向链接边表复制到$\mathcal{D}$中
2. 应用第 2.1 节介绍的平面扫描算法，计算出S_1的各边与S_2的各边之间的全部交点
   在每个事件点处，除了对$\mathcal{T}$和$\mathcal{Q}$所必需的操作外，还要完成如下工作：
   (i) 若该事件点同时涉及到S_1和S_2中的边，则按照上面介绍的方法对$\mathcal{D}$做更新
   （本节中所介绍的，只是S_1的一条边穿过S_2的一个顶点的情况）
   (ii) 找到紧邻于该事件点左侧的那条半边，将它存在于$\mathcal{D}$中对应于该点的顶点处
3. (* 现在，$\mathcal{D}$已经基本上可以称作是对应于$\mathcal{O}(S_1, S_2)$的双向链接边表 *)
   (* 只是各张面的信息还有待计算出来 *)
4. 通过对$\mathcal{D}$的遍历，找出$O(S_1, S_2)$中的所有边界环
5. 构造一张图$\mathcal{G}$：
   其中的每个节点，分别对应于一个边界环；
   其中的每条弧，都联接了一个空洞环与紧邻于该环最左端顶点左侧的另一个环
   找出$\mathcal{G}$中所有的连通子块
   (* 用以确定$\mathcal{G}$中各条弧的信息，在上面第 2 行的第 2 项中已经计算出来了 *)
6. **for** ($\mathcal{G}$的每一连通子块)
7. **do** 令 C 为该连通子块（唯一）的外边界环；由此环围成的面，记作 f
   生成对应于 f 的一个面记录
   将 OuterComponent()指向 C 的某条半边
   生成一个 InnerComponents()列表
   (* 对应于该连通子块中的每一个孔洞环， *)
   (* 在该列表中都有一个指针，指向对应的空洞环上的某条边 *)
   将各（内、外）环上所有半边的 IncidentFace()指针，指向对应于 f 的面记录
8. 按照上面介绍的方法，
   给$O(S_1, S_2)$中的每张面都做上标记，
   指明各张面在S_1和S_2中分别被包含于哪张面中

〖定理 2.6〗

给定任意两个平面子区域划分 S_1 和 S_2，其复杂度分别为 n_1 和 n_2，令 $n = n_1 + n_2$。则可以在 $O(n\log n + k\log n)$时间内计算出 S_1 与 S_2 的叠合，其中 k 为叠合结果的复杂度。

〖证明〗

（按照上面的算法，）第 1 行复制双向链接边表的计算只需$O(n)$时间。根据〖引理 2.3〗，第 2 行的平面扫描需要$O(n\log n + k\log n)$。第 4~7 行的工作是填写面记录，需要的时间线性正比于$O(S_1, S_2)$的复杂度。（只需一次深度优先搜索，即可确定所有的连通子块。）最后一步，是在得出的子区域划分中，给各张面做上标记，指明它在原先的两个子区域划分中分别包含于哪张面中，这一步也可以在$O(n\log n + k\log n)$时间内完成。 ❑

2.4 布尔运算

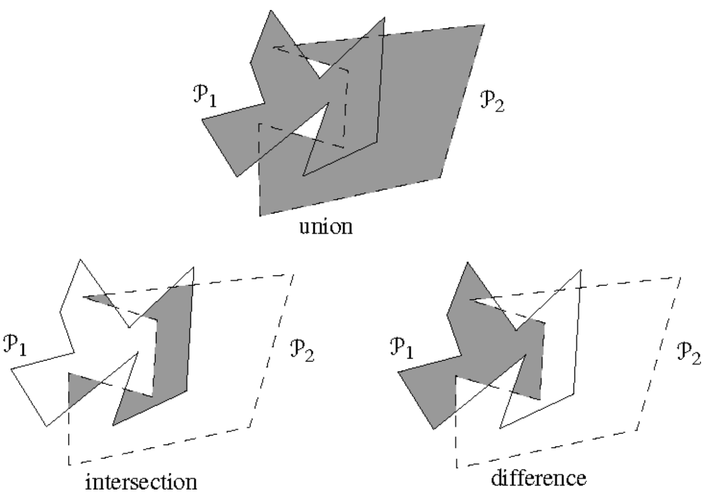


图2-34 两个多边形P_1和P_2的布尔运算（boolean operation）——并（union）、交（intersection）和差（difference）

地图叠合算法是一个强有力的工具，在众多不同的应用中它都可以大显身手。其中特别有用的一点，就是可以用以实现两个多边形P_1 和P_2 的布尔运算（boolean operation）——并（union）、交（intersection）和差（difference）。图 2-34 给出了这样一个例子。请注意，经过运算后，结果可能已经不止是一个多边形。可以出现多个多边形子区域，其中甚至可能存在空洞。

为了实现布尔运算，我们把多边形看作平面地图，其边界面分别标为 P_1 和 P_2。首先计算这两幅地图的叠合，然后根据我们所要进行的布尔运算种类，从叠合的结果中抽取出一些做有相应标记的面。比如，要是计算 P_1 和 P_2 的交（$P_1 \cap P_2$），则从叠合结果中抽取出来的，就是所有同时标有 P_1 和 P_2 标记的那些面。而要是计算 P_1 和 P_2 的并（$P_1 \cup P_2$），则从叠合结果中抽取出来的，就是所有标

有 P_1 或 P_2 标记的那些面。再如，要是计算 P_1 和 P_2 的差（$P_1 \backslash P_2$），则从叠合结果中抽取出来的，就是所有标有 P_1 标记、却没有 P_2 标记的那些面。

任意给定 P_1 的一条边与 P_2 的一条边，若它们相交，则交点肯定是 $P_1 \cap P_2$ 的一个顶点。因此，（求交运算）算法的运行时间为 $O(n\log n + k\log n)$，其中，n 为 P_1 和 P_2 中的顶点总数，而 k 为 $P_1 \cap P_2$ 的复杂度。对于其它类型的布尔运算，这一点也是一样的——无论进行的是何种运算，任何两条边的交点，都是最终结果中的一个顶点。这样，就立即可以得出如下结论：

〖推论 2.7〗

任意给定两个多边形 P_1 和 P_2，设其顶点数分别为 n_1、n_2，令 $n := n_1 + n_2$。则可以在 $O(n\log n + k\log n)$ 时间内，计算出 $P_1 \cap P_2$、$P_1 \cup P_2$ 或 $P_1 \backslash P_2$，其中 k 为最终输出的复杂度。

2.5 注释及评论

线段求交是计算几何（computational geometry）中的基本问题之一。本章介绍的 $O(n\log n + k\log n)$ 算法，是由Bentley和Ottmann[47]在 1979 年提出的——在此前的若干年，Shamos和Hoey [351]已经解决了（线段交点的）检测问题，这个问题关心的是，（一组线段之间）是否存在至少一个交点，他们的算法需要 $O(n\log n)$ 时间。本章还介绍了一种方法，将空间复杂度从 $O(n+k)$ 降低到 $O(n)$，这个算法是由Pach和Sharir[312]提出的。Brown[77]提出了另一种方法，也可以实现这样的改进。

“报告各线段之间所有交点”这一问题的下界（lower bound）为 $\Omega(n\log n + k)$，因此，当k比较大的时候，本章介绍的算法并不是最优的。在构造最优算法的过程中，第一步是由Chazelle[88]完成的，他给出了一个算法，时间复杂度为 $O(n\log^2 n/\log\log n + k)$。1988 年，Chazelle和Edelsbrunner[99][100]首次提出了一个 $O(n\log n + k)$ 的算法。遗憾的是，他们的算法需要占用 $O(n+k)$ 的存储空间。后来，Clarkson和Shor[133]，以及Mulmuley[288]分别给出了各自的随机增量式算法，其期望运行时间（expected running time）都是 $O(n\log n + k)$。（关于随机算法，可以参见第 4 章的详细介绍。）这两个随机算法（randomized algorithm)空间复杂度分别为 $O(n)$ 和 $O(n+k)$。与Chazelle和Edelsbrunner的算法不同的是，这些随机算法还适用于计算一组曲线的交点。最近，Balaban[35]针对线段求交问题，给出了一个确定性算法（deterministic algorithm）。这个算法的时间复杂度为 $O(n\log n + k)$ 时间，空间复杂度为 $O(n)$。因此，这就成为了同时在时间、空间上都达到最优的第一个算法。而且，这个算法也适用于曲线。

与一般情况的线段求交问题相比，有些类型的线段求交问题更加容易。这样的一个例子就是：所有线段被划分为红、蓝两个集合，在每个集合内部，线段互不相交。（实际上，这恰好正是本章所讨论的网络叠合问题。然而在这里所介绍算法中，“同一集合内部的线段互不相交”这一条件并未用到。）这种所谓的红-蓝线段求交问题（red-blue line segment intersection problem）,早在一般性

线段求交问题的最优算法被提出之前，就已经由Mairson和Stolfi[262]解决了，他们的算法时间复杂度为O(n**log**n + k)，空间复杂度为O(n)。Chazelle等人 [101]，以及Palazzi和Snoeyink [315]，也针对红-蓝线段求交问题，独立地提出了各自的最优算法。要是两个线段集各自构成一个连通的子区域划分，情况就更好了——正如Finke和Hinrichs[176]所证明的，在这种情况下，两个子区域划分的叠合可以在O(n+k)时间内计算出来。这个结果，对此前Nievergelt和Preparata[293]、Guibas和Seidel[200]以及Mairson和Stolfi[262]针对地图叠合问题所得出的成果，做了一般化推广与改进。

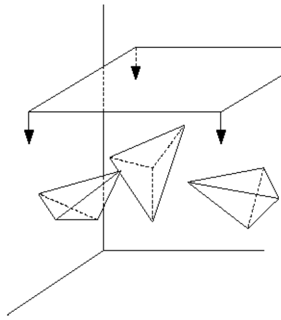


图2-35 空间扫描算法

平面扫描是几何算法设计中最有用的经典算法模式之一。在计算几何领域，Shamos和Hoey[351]、Lee和Preparata[250]以及Bentley和Ottmann[47]各自独立地率先提出了基于这一模式的算法。平面扫描算法尤其适用于对一组物体求交，此外，在求解许多其它问题时，也可采用这一模式。第 3 章将利用平面扫描算法，解决多边形三角剖分问题的一部分；而在第 7 章我们将看到另一个平面扫描算法，它可以计算出点集的Voronoi图。本章的算法使用了一条水平线自上而下扫过整个平面。而对另一些问题，以其它的方式扫描平面将更加方便。例如，可以使用一条旋转的直线来扫描平面（第 15 章将给出这样的一个例子）；也可以使用一条伪直线——虽然这条“直”线不见得是直的，其行为却差不多相当于一条直线 [159]。在高维空间中，平面扫描技术也大有用武之地。当然，如 图 2-35 所示，此时我们是用超平面（hyperplane）扫过整个（多维）空间 [213][311][324]——因此，这类算法也称为空间扫描算法（space sweep algorithm）。

本章介绍了一种用来存储子区域划分的数据结构——双向链接边表。当然，可以用来存储子区域划分的数据结构还有很多种，比如Baumgart[40]提出的有翼边结构（winged edge structure），以及Guibas和Stolfi[202]提出的四叉边结构（quad edge structure）。种种此类数据结构，都大同小异。它们所支持的功能大多雷同，只不过有些结构可以用更少的字节来存储每条边。

2.6 习题

习题 2.1 设S为由n条互不相交的线段构成的一个集合，其中每条线段的上端点都落在直线y = 1 上，而下端点都落在直线y = 0 上。如 图 2-36 所示，由于引入了这些线段，条带区

域$[-\infty:\infty]\times[0:1]$被分成了$n+1$个子区域。

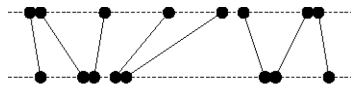


图2-36 横跨于水平条带之间的一组线段

试给出一个$O(n\log n)$的算法，通过将 S 中的线段组织成一棵二分查找树，使得在任意给定一个待查询点（query point）后，都能够在$O(\log n)$时间内，找到它所在的那个子区域。此外，也请详细地给出相应的查找算法。

习题 2.2　所谓的相交检测问题（intersection detection problem）指的是：对于由任意n条线段构成的集合S，判断S中是否有两条线段相交[①]。试给出一个平面扫描算法，在$O(n\log n)$时间内解答这类相交检测问题。

习题 2.3　试修改算法 FINDINTERSECTIONS（及其调用的子函数）的代码，使其消耗的空间规模从$O(n+k)$降至$O(n)$。

习题 2.4　设 S 为由平面上 n 条线段构成的一个集合，其中的线段可能会（部分地）相互叠合。比如，S 中可能同时包含线段(0, 0)(1, 0)和线段(-1, 0)(2, 0)。现在，我们希望计算出 S 中所有的交点。更准确地说，我们希望找出 S 中所有线段对之间的真交点（proper intersection，即两条不平行线段之间的交点），并且对每条线段的各个端点，找出经过它的所有线段。为此，你可以借鉴算法 FINDINTERSECTIONS。

习题 2.5　下列等式中，哪些是必然成立的？

$$\mathrm{Twin}(\mathrm{Twin}(\vec{e})) = \vec{e}$$

$$\mathrm{Next}(\mathrm{Prev}(\vec{e})) = \vec{e}$$

$$\mathrm{Twin}(\mathrm{Prev}(\mathrm{Twin}(\vec{e}))) = \mathrm{Next}(\vec{e})$$

$$\mathrm{IncidentFace}(\vec{e}) = \mathrm{IncidentFace}(\mathrm{Next}(\vec{e}))$$

习题 2.6　试给出一个双向链接边表的实例，对其中的一条边 e 而言，IncidentFace($\vec{e}$)和IncidentFace(Twin($\vec{e}$))是同一张面。

习题 2.7　试给出一个子区域划分的实例，在与之对应的双向链接边表中，对于每一条半边$\vec{e}$，都有 Twin($\vec{e}$) = Next($\vec{e}$)成立。具有这一性质的子区域划分，最多可能包含多少张面？

习题 2.8　试以伪代码的形式给出一个算法，对任一顶点 v，在双向链接边表中找出与之相关联的所有顶点。此外，也请以伪代码的形式给出另一个算法，在子区域划分中找出参与围成某张面的所有边（注意，这里的子区域划分可以是不连通的）。

[①] 具体是哪两条线段，这里并不关心。——译者

习题 2.9　假设给定了一个连通子区域划分所对应的双向链接边表结构。试以伪代码的形式给出一个算法，找出至少有一个顶点落在外边界上的所有面。

习题 2.10　设 S 是复杂度为 n 的一个子区域划分，而 P 为由 m 个点构成的一个集合。试给出一个平面扫描算法，对于 P 中的每一个点，找出包含它的那张面。试证明，你的算法的运行时间为 $O((n+m)\log(n+m))$。

习题 2.11　设 S 为由平面上 n 个圆环构成的一个集合。试给出一个平面扫描算法，计算出这些圆环之间的所有交点。（这里处理的是圆环（circle）而不是圆盘（disc），因此若某个圆环完全落在另一个圆环的内部，我们并不认为它们相交。）你给出的算法必须在 $O((n+k)\log n)$时间内运行结束，其中 k 为实际的交点数目。

习题 2.12　设 S 为由平面上 n 个三角形构成的一个集合。这些三角形的边界都是互不相交的，但某个三角形有可能完全包含在另一个三角形的内部。给定由平面上 n 个点组成的一个集合 P。试给出一个 $O(n\log n)$的算法，在 P 中找出不属于任何三角形内部的所有点。

习题 2.13*　如图 2-37 所示，设S为由平面上n个互不相交的三角形构成的一个集合。我们希望找出具有下列性质的一组共n - 1 条线段：

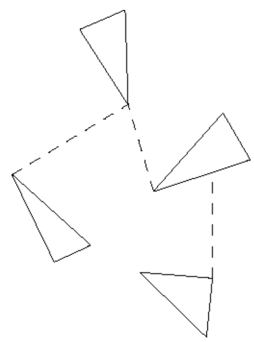


图2-37 平面上一组互不相交的三角形

其中的任何一条线段，都联接于分别来自不同三角形边界的两个点之间；

任何两条线段的内部都不相交，而且各条线段的内部也不会与任何三角形相交；

这些线段共同将所有三角形都联接起来——也就是说，沿着这些线段以及三角形的边界，可以从任何一个三角形到达另一个三角形。

请设计一个平面扫描算法，在 $O(n\log n)$时间内解决这一问题。必须明确地说明你定义了哪些事件，使用了哪些数据结构；而且，还要说明可能出现的所有情况，以及对应的处理方法。此外，你还要指出在平面扫描过程中的不变量（或不变性）。

习题 2.14　设 S 为由平面上 n 条互不相交的线段构成的一个集合，而 p 为一个不属于 S 中任何线段的一个点。

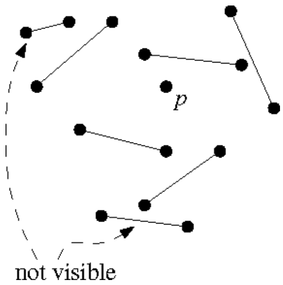


图2-38 平面上一组互不相交的线段

我们的任务是，在 S 中找出所有与 p 可见的线段——换而言之，在任何一条这样的线段上，都存在某个点 q，使得开线段 $\overline{pq}$ 不与 S 中的任何线段相交。试给出一个 *O*(n**log**n)的算法来解决这个问题（通过旋转以 p 为端点的一条射线）。

3

多边形三角剖分：画廊看守

面对出自名家手笔的绘画作品，怦然心动的可不止是艺术爱好者，罪犯们也是如此。这类作品价值不菲、易于运输，而且很显然，不愁出不了手。正因为此，艺术画廊都必须对其拥有的作品严加看管。白天，可以由值班人员担负起看守的任务，然而到了晚上，这项任务就落到了摄像机的肩上。通常，这些摄像机都被挂在天花板上，绕着某个垂直的轴旋转。由摄像机采集到的图像，将被传送到守夜值班室的电视屏幕上。显然，眼睛同时要盯住的屏幕数量越少，守夜员就可以更轻松一些，因此，总是希望能够尽可能地减少摄像机的数目。摄像机数目更少的另外一个好处还在于，相应的保安系统可以成本更低。另一方面，摄像机的数目也不可能过少——因为，画廊内的每一个角

落，都必须被落在至少一台摄像机的视野之内。

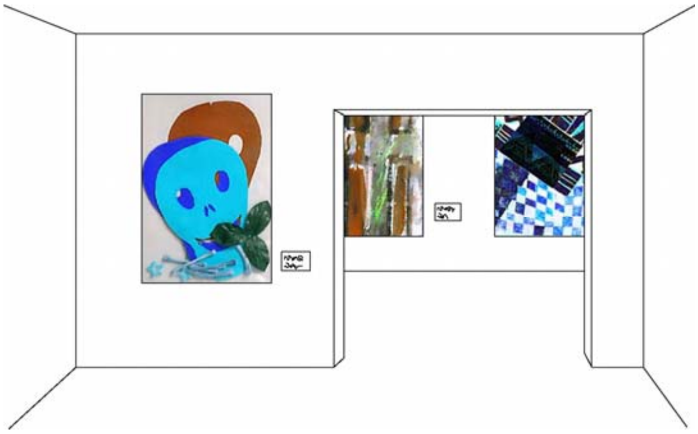


图3-1 艺术画廊

因此，摄像机的安装位置很有讲究，我们的目标是：使每台摄像机都能在画廊中照应到更大的范围。这样，就导出了通常所谓的艺术画廊问题（art gallery problem）：如 图 3-2 所示，给定一个画廊，需要多少台摄像机？应该将它们分别安装在什么位置？

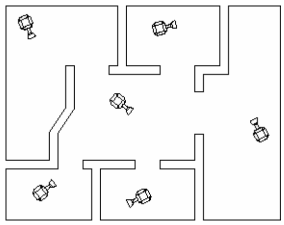


图3-2 监视画廊的一组摄像机

3.1 看守与三角剖分

为了更确切地对艺术画廊问题做一定义，必须首先将画廊的概念做形式化处理。自然地，每个画廊都是一个三维空间，然而通过它的平面结构图，我们就可以获得足够的信息来确定摄像机的安放位置。因此，可以利用平面多边形的模型来表示一个画廊。我们还进一步做出限制，要求画廊的模型应是简单多边形（simple polygon）——即由单个不自交的、封闭的多边形链所围出的区域[①]。这样，就不允许出现空洞。一台摄像机在画廊中的位置，对应于多边形中的一个点。如 图 3-3 所示，对于多边形内部的任何一点，只要联接于它与某台摄像机之间的开线段完全落在多边形的内部，它就能被这台摄像机监视到。

① 更准确而简洁地，简单多边形的边界是一条约当曲线（Jordan curve）。——译者

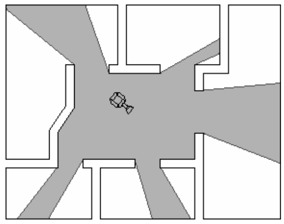


图3-3 单台摄像机所能看守的区域

为了看守一个简单多边形，需要多少台摄像机呢？显然，这要取决于具体的多边形：多边形越是复杂，需要的摄像机就越多。因此，我们将根据多边形的顶点数目n，来界定所需摄像机的数量。然而，即使是顶点数目相等的两个多边形，其中的一个，也可能比另一个更容易看守。为了保险起见，我们所考虑的将是最坏的情况——也就是说，将要给出的只是一个上界（upper bound），该上界适用于由n个顶点组成的所有简单多边形。（如果能够为特定的某个多边形，找到其所需摄像机的最小数目，而不是一个笼统的最坏上限，那自然是再好不过的了。但不幸的是，“计算出特定多边形所需摄像机的最小数目”这一问题，是NP-难的（NP-hard）[①]。）

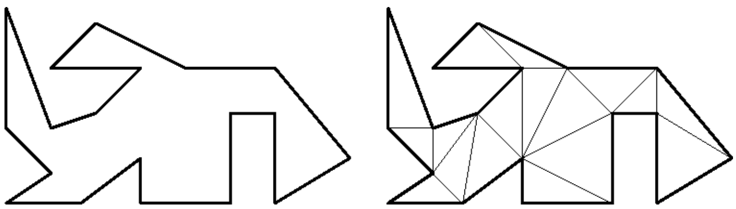


图3-4 一个简单多边形，及其可能的一个三角剖分

设$\mathcal{P}$为包含n个顶点的简单多边形。在确定为看守$\mathcal{P}$所需摄像机的最小数目时，由于$\mathcal{P}$的形状可能极为复杂，所以我们似乎无从下手。首先将$\mathcal{P}$分解为很多块，每一块都很容易看守——具体而言，这里的“块”就是三角形。为了完成这种分解，需要添加一些对角线（diagonal），将某些顶点对联接起来。所谓对角线是一条开的线段，它联接于$\mathcal{P}$的某两个顶点之间，而且完全落在$\mathcal{P}$的内部[②]。通过极大的[③]一组互不相交的对角线，可将一个多边形分解为多个三角形——称作该多边形的三角剖分（triangulation），参见 图 3-4。（由于这一互不相交的对角线集合必须满足最大化的条件，故可以保证任何三角形各边的内部，都不包含多边形的任何顶点。如果多边形中存在三个共线的顶点，就可能会出现这种情况。）通常，简单多边形的三角剖分不是唯一的。例如 图 3-4 所示的这个多边形，就有多种不同的三角剖分方案。给定$\mathcal{P}$的一个三角剖分$\mathcal{T}_{\mathcal{P}}$，只要在（由此确定的）每个三角形中放置一台摄像机，就可以实现对整个多边形的看守。然而，是否每个简单多边形总存在一个三角剖分呢？

[①] 参见[246]。——译者

[②] 请注意：根据这一定义，多边形的边不是对角线，而且对角线不能通过任何顶点，更不能与任何边有局部重合。——译者

[③] 所谓“极大”，指的是再增加任何一条对角线，都会与其中原有的某条对角线相交。——译者

如果存在，其中三角形的数目又是多少呢？下面这则定理回答了这些问题。

〖定理 3.1〗

任何简单多边形都存在（至少）一个三角剖分；若其顶点数目为 n，则它的每个三角剖分都恰好包含 n - 2 个三角形。

〖证明〗

我们通过对 n 进行归纳来证明。若 n = 3，则多边形本身就是一个三角形，此时是定理的平凡形式，显然成立。现假设 n > 3，而且该定理对所有 m < n 都成立。任取一个有 n 个顶点的简单多边形 P。我们将首先证明，在 P 中存在（至少）一条对角线。

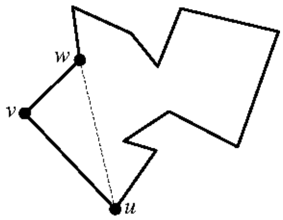


图3-5 情况1：线段$\overline{uw}$完全落在P的内部

令v为在P中最靠左的顶点（如果这种顶点有多个，只选用其中的最低者）。沿P的边界，考虑与v相邻的那两个顶点，分别记为u和w。若开线段$\overline{uw}$完全落在P的内部（如图 3-5 所示），则它就是一条对角线。

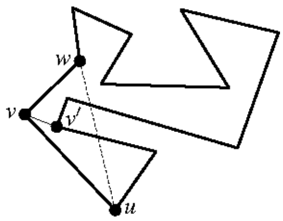


图3-6 情况2：线段$\overline{uw}$不完全落在P的内部

否则，如图 3-6 所示，在由u、v和w所确定的三角形内部，必然存在至少一个顶点（当然，它也可能正好落在开线段[①]$\overline{uw}$上）。在这些顶点中，令v'为与$\overline{uw}$相距最远者。现在，联接v'和v的那条（开）线段，不可能与P的任何边相交——否则，这条边的两个端点中，必有其一会落在

① 此处原文为“对角线”。与前面的定义不合。——译者

这个三角形内部，而且该端点到$\overline{uw}$的距离要比v'更远。因此，$\overline{vv'}$就是一条对角线[①]。

既然对角线必定存在，那么这样一条对角线就会将 P 切分为两个子多边形 P_1 和 P_2。分别记 P_1、P_2 的顶点数目为 m_1、m_2，则 m_1 和 m_2 都小于 n。这样，根据归纳假设，P_1 和 P_2 都能够被三角化。于是，P 也能够被三角化。

现在只需要再说明：P 的任何三角剖分都必然由 n - 2 个三角形构成。为此，任取 T_P 中的一条对角线。这条对角线将 P 切分为两个子多边形，设它们各有 m_1、m_2 个顶点。除了该对角线的两个端点之外，其它的每个顶点只归属于这两个子多边形中的某一个。于是就有 $m_1 + m_2 = n + 2$。根据归纳假设，P_i 的任何三角剖分都由 $m_i - 2$ 个三角形组成（i = 1 或 2），故 T_P 包含的三角形数目为 $(m_1 - 2) + (m_2 - 2) = n - 2$。 ❑

由〖定理 3.1〗可得出推论：包含n个顶点的任一简单多边形，都可用n - 2 台摄像机来看守。然而，为每个三角形配备一台摄像机，不免有些浪费。比如，只要将一台摄像机安装在任何一条对角线上，就可以看守住（与该对角线关联的）两个三角形——也就是说，如果精心挑选出若干对角线，然后在那些位置安装摄像机，就可能将所需摄像机的总数减少到大约n/2。而似乎更好的策略是，将摄像机安装在（多边形的）顶点上——毕竟，一个顶点可能同时与更多的三角形相关联，这样，只需一台摄像机，就可以将与之相关联的所有三角形都看守住。这样，就导出了下面的方法。

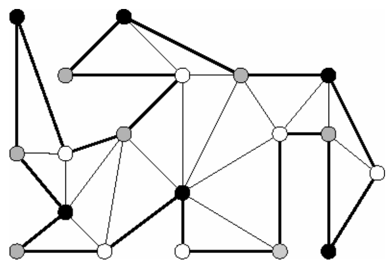


图3-7 根据三角剖分对顶点进行3-染色

令T_P为P的一个三角剖分。选出P的部分顶点组成一个子集，使得T_P中的每个三角形，都有至少一个顶点来自于该子集；然后，在被挑选出的每个顶点处，分别放置一台摄像机。为了找出这样一个子集，可以使用白、灰和黑三种颜色，给P的所有顶点染色（如 图 3-7 所示）。我们的染色方案必须满足：由任何边或者对角线联接的两个顶点，所染的颜色不能相同——称作“对经过三角剖分后的多边形的 3-染色（3-coloring）”。三角剖分后的多边形经过如此染色，其中每个三角形都有（且仅有）一个白色、灰色和黑色的顶点。因此，只要在同色（比如灰色）的各顶点处分别放置一台摄像机，就必然可以看守整个多边形。进一步地，若选用点数最少的那一类同色顶点，并为它们配备摄像机，则只需不超过$\lfloor \frac{n}{3} \rfloor$台摄像机，即可看守住$P$。

[①] 更严密地，还应该说明“开线段 $\overline{vv'}$ 完全落在多边形内部”。这可以反证——否则，$\overline{vv'}$ 必然与 P 的某条边相交。——译者

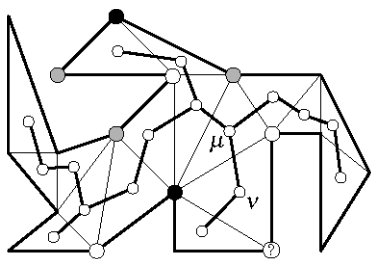


图3-8 3-染色方案必然存在

然而，3-染色方案是否总是存在？答案是肯定的。为了理解这一结论，让我们来看看所谓“$\mathcal{T}_P$的对偶图”——记之为$\mathcal{G}(\mathcal{T}_P)$。对应于$\mathcal{T}_P$中的每个三角形，$\mathcal{G}(\mathcal{T}_P)$都有一个顶点。将对应于顶点 v 的三角形记作 t(v)。若 t(v)与 t(u)共用一条对角线，则在 v 和 u 之间就设置一条弧。这样，$\mathcal{G}(\mathcal{T}_P)$中的各条弧就分别对应于$\mathcal{T}_P$中的各条对角线。任何一条对角线都会将$\mathcal{P}$一分为二，故移去$\mathcal{G}(\mathcal{T}_P)$的任意一条弧，$\mathcal{G}(\mathcal{T}_P)$都会分裂成两个（各自连通的）部分。因此，$\mathcal{G}(\mathcal{T}_P)$必然是一棵树——当然，如果允许多边形内含有空洞，这个结论就不一定成立）。这样，只要对该图进行一次（比如，深度优先）遍历（traverse），就可以得到一种 3-染色的方案。以下介绍具体的做法。在深度优先遍历的过程中，始终都保证这样一点：已经访问过的三角形的所有顶点，都已被染上了白色、灰色或黑色；而且，任何一对（通过对角线或边）相互联接的顶点，颜色互异。由此可以保证：在访问完所有的三角形之后，可得到一个 3-染色的方案。深度优先遍历可从$\mathcal{G}(\mathcal{T}_P)$的任一顶点开始；第一个被访问的三角形，其三个顶点将分别被染上白色、灰色或黑色（次序无所谓）。现在，假设从$\mathcal{G}$的一个顶点 u 到达另一个顶点 v。既然如此，t(v)和 t(u)之间肯定存在一条公共对角线。由于 t(u)的三个顶点都已经被染上了互异的颜色，所以 t(v)的三个顶点中只有一个顶点需要染色。而且，只有一种颜色可供它使用——准确地，就是 t(v)与 t(u)之间公共对角线所没有用到的那种颜色。$\mathcal{G}(\mathcal{T}_P)$是一棵树，故在此时，与 v 相邻（除 u 之外）的其它顶点都尚未访问到，因此的确可以将剩下的这一颜色赋给这个顶点。

总而言之，对于经过三角剖分的任意简单多边形，都能够（对其顶点）实施 3-染色。于是，只需$\lfloor \frac{n}{3} \rfloor$台摄像机，就可以看守住任何一个（包含n个顶点的） 简单多边形。不过，我们的成本还可能更低。毕竟，放置在顶点处的一台摄像机，其能够看守的范围，可能不止是与之相关联的那些三角形。然而不幸的是，对任何$n \geq 3$，都存在一个（包含n个顶点的）简单多边形，它的确需要$\lfloor \frac{n}{3} \rfloor$台摄像机。这样的一个例子就是所谓的“梳状多边形”（comb-shaped polygon）：如图 3-9 所示，它有一条长长的水平基边，以及$\lfloor \frac{n}{3} \rfloor$个分别由两条边形成的“梳齿”。任何两个相邻的梳齿之间，由一条水平边相联。只要适当地安排各顶点的位置，就总能够保证：单台摄像机无论放置在多边形内的什么位置，都不可能同时看到两个梳齿。因此，我们不能指望能够依靠某种策略，每次都找到少于$\lfloor \frac{n}{3} \rfloor$台摄像机。换而言之，就最坏情况来而言，上述 3-染色的方法已经是最优的了。

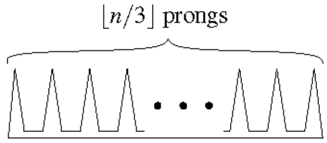


图3-9 梳状n边形需要$\lfloor \frac{n}{3} \rfloor$台摄像机

以上就证明了组合几何学（combinatorial geometry）的一个经典结果：

〖定理 3.2（艺术画廊定理）〗

包含 n 个顶点的任何简单多边形，只需（放置在适当位置的）$\lfloor \frac{n}{3} \rfloor$台摄像机就能保证：其中任何一点都可见于至少一台摄像机。有的时候，的确需要这样多台摄像机。

现在我们已经知道，$\lfloor \frac{n}{3} \rfloor$台摄像机总是够用的。然而，我们还没有有效的算法，以计算出各台摄像机的具体位置。为此，需要一个快速的算法，以实现对任何简单多边形的三角剖分。同时，通过该算法，还应该能够导出一个合理的数据结构（比方说，双向链接边表），来表示三角剖分后的结果——这样，（在遍历时）只需常数时间，就可以从一个三角形转到它的一个邻居。一旦已经得到了这种形式的结构表示，就可以在线性时间内，按照上述方法——深度优先遍历对偶图，完成 3-染色，按照颜色将所有顶点分为三类，取出数量最少的一类顶点，并在这类顶点处放置摄像机——确定总数不超过$\lfloor \frac{n}{3} \rfloor$台摄像机的具体位置。接下来的一节，将介绍如何在 $O(n\log n)$时间内构造一个三角剖分。提前借用这一结果，就可以得出下面有关多边形看守的最后结论：

〖定理 3.3〗

任给一个包含 n 个顶点的简单多边形 P。总可以在 $O(n\log n)$时间内，在 P 中确定$\lfloor \frac{n}{3} \rfloor$台摄像机的位置，使得 P 中的任何一点都可见于其中的至少一台摄像机。

3.2 多边形的单调块划分

任给一个包含n个顶点的简单多边形P。根据〖定理 3.1〗，P的三角剖分总是存在。那个定理的证明本身就是构造式的，故马上就可以由此导出一个递归的三角剖分算法：找到一条对角线，将原多边形切分为两个子多边形，然后递归地对两个子多边形实施三角剖分。为了找到这样一条对角线，我们找出P中最靠左的顶点v，然后试着将与v相邻的两个顶点u和w联接起来；如果不能直接联接这两个顶点，就在由u、v和w确定的三角形内，找出距离$\overline{uw}$最远的那个顶点，然后将它与v联接起来。按照这种方法，需要花费线性的时间才能找到一条对角线。而且，（在最坏情况下）这条对角线将P

切分为一个三角形，以及一个含有n-1 个顶点的多边形。我们的确可能一直都是联接u和w——这就是最坏情况。在这种最坏情况下，上述三角剖分算法需要运行平方量级的时间。能否更快呢？对于某些类型的多边形，的确可以更快。

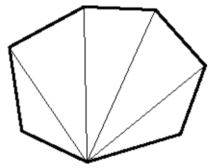


图3-10 凸多边形的三角剖分可以在线性时间内构造出来

比如凸多边形（convex polygon）就很容易：如 图 3-10 所示，取出多边形的任何一个顶点，除了它的两个邻居之外，在这个顶点与其它的所有顶点之间分别联接一条对角线。整个过程只需要线性的时间。因此，对非凸多边形进行三角剖分的一种可能的方法就是：首先将Ꝑ划分为多个凸块，然后分别对每块做三角剖分。然而不幸的是，将多边形划分为凸块的难度，与对它做三角剖分是一样的 [①]。因此，我们将把Ꝑ划分为所谓的“单调块”（monotone piece）——这项工作要容易得多。

一个简单多边形称作“关于某条直线 l 单调”（monotone with respect to a line l），如果对任何一条垂直于 l 的直线 l'，l'与该多边形的交都是连通的。换而言之，它们的交或者是一条线段，或者是一个点，也可能是空集。

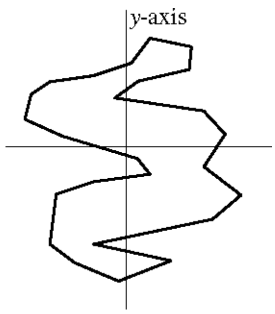


图3-11 单调多边形（monotone polygon）

如果一个多边形关于y坐标轴单调（图 3-11），则称它是y-单调的（y-monotone）。下面这个性质，是y-单调多边形（y monotone polygon）的一个特征：在沿着多边形的左（右）边界，从最高顶点走向最低顶点的过程中，我们始终都是朝下方（或者水平）运动，而绝不会向上。

我们对多边形Ꝑ进行三角剖分的策略是：首先将Ꝑ划分成若干个y-单调块，然后再对每块分别进行三角剖分。可以按照下面的方法，将一个多边形划分成单调块。设想我们沿着Ꝑ的左或右边界，从其最高顶点走向最低顶点。在某些顶点处，我们的行进方向可能会从向下转成向上，或者从向上转

[①] 比如前面所提到的梳状多边形，对它的任何凸分解，都需要划分出 $\lfloor \frac{n}{3} \rfloor = \Omega(n)$ 个三角形。——译者

成向下——这些位置称作拐点（turn vertex）。为了将Ꝓ划分成多个y-单调块，就必须消除这些拐点。为此可以引入对角线。如 图 3-12 所示，若在某个拐点v处，与之关联的两条边都朝下 ①，而且在此局部，多边形的内部位于v的上方，那么就必须构造一条从v出发、向上联接的对角线。

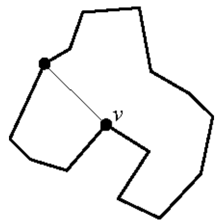


图3-12 通过引入对角线消除拐点

这条对角线将原多边形一分为二，而且在划分出来的两块中，顶点v都会出现。此外，在其中的任何一块中，与v相关联的两条边，必然有一条朝下（具体讲，就是从原多边形中继承下来的那条边），而另一条则朝上（也就是所引入的对角线）②。也就是说，在两个子块中，v都不再是一个拐点。如果与v相关联的两条边都朝上，而且在此局部，多边形的内部位于v的下方 ③，那么就需要构造一条从v出发、向下联接的对角线。显然，拐点有多种不同类型，故需要更加准确地加以区分。

为了更加仔细地对不同类型的拐点做出定义，需要特别注意那些 y-坐标相同的顶点。为此，要定义好“下方”和“上方”的概念：所谓“点 p 处于点 q 的下方”，是指 $p_y < q_y$，或者 $p_y = q_y$ 而 $p_x > q_x$；而所谓“点 p 处于点 q 的上方”，是指 $p_y > q_y$，或者 $p_y = q_y$ 而 $p_x < q_x$。（你可以想象着相对于原来的坐标系，沿顺时针方向，将整个平面旋转“一丁点”——这样，任何两个点都不会具有相同的 y-坐标，而且上面所定义的上/下关系，在旋转后的平面上依然保持不变。）

Ꝓ的顶点可划分为五类（参见 图 3-13）。其中四类都是拐点：起始顶点、分裂顶点、终止顶点以及汇合顶点。它们的定义如下。顶点v是一个起始顶点（start vertex），如果与它相邻的两个顶点的高度都比它低，而且在v处的内角小于π；如果该内角大于π，v就是一个分裂顶点（split vertex）。（注意，既然与v相邻的两个顶点都比v更低，此处的内角就不可能等于π。）顶点v是一个终止顶点（end vertex），如果与它相邻的两个顶点的高度都比它高，而且在v处的内角小于π；如果该内角大于π，v就是一个汇合顶点（merge vertex）。这四类拐点以外的所有顶点，都是普通顶点（regular vertex）。也就是说，在每个普通顶点的两个相邻顶点中，必然有一个比它高，而另一个则比它低。之所以要

---

① 由于边是没有方向的，故准确地讲，应该是“与 v 相邻的两个顶点，y-坐标均低于 v”。（后面的“朝上”也有这个问题。）如果再加上“在此局部多边形的内部位于其上方”的条件，则这种顶点也被称作石笋（stalagmite）。——译者

② 再次地，由于这里并没有定义边的方向，故准确的描述应该是：与 v 相邻的两个顶点中，必然有一个（的 y-坐标）低于 v（该顶点与 v 之间的边，来自原多边形），而另一个要高于 v（该顶点与 v 是所引入对角线的两个端点）。——译者

③ 这种顶点也别称作钟乳石（stalactite）。——译者

给不同类型的顶点取这样的名字，是因为我们的算法要进行一次自上而下的平面扫描，在此过程中，要维护扫描线与多边形的交集。当扫描线触及一个分裂顶点时，交集中的某个（连通的）部分就要分裂；当扫描线触及一个汇合顶点时，则有两个（连通的）部分会汇合起来；诸如此类。

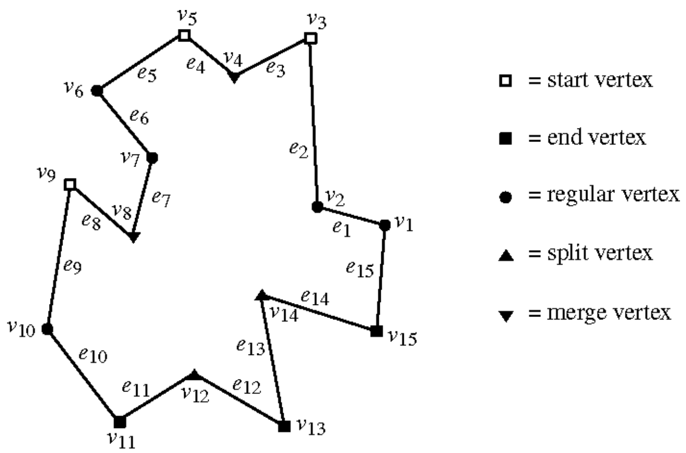


图3-13 五种类型的顶点

多边形中局部的非单调性，正来自于这些分裂顶点和汇合顶点。而且反过来，下面这个命题看似更强，却也竟然是成立的：

〖引理 3.4〗

一个多边形若既不含分裂顶点，也不含汇合顶点，则必然是 y-单调的。

〖证明〗

假设 $\mathcal{P}$ 不是 y-单调的。我们来证明，$\mathcal{P}$ 中必然含有一个分裂顶点，或者一个汇合顶点。

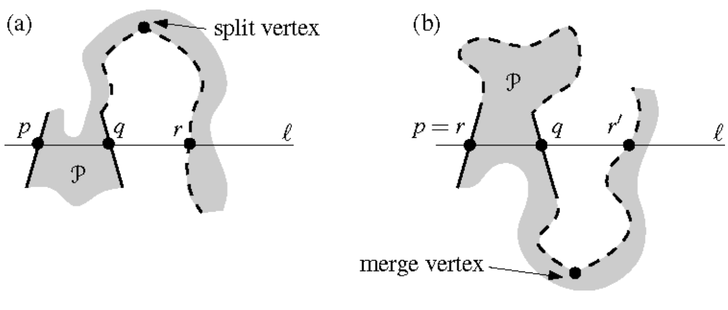


图3-14 〖引理3.4〗的证明中所涉及到的两种情况

既然$\mathcal{P}$不单调，则根据定义必然存在某条水平线l，它与$\mathcal{P}$的交集含有（至少）两个（各自）连通子集。只要选取得当，总能找到一条这样的l：l中最左边的那个（连通）子集是一条线段，而不是一个点。分别令p和q为该线段的左、右端点。现在，从q开始，沿着$\mathcal{P}$的边界行进——行进的方向要使得$\mathcal{P}$（在任何局部都）居于左侧（也就是说，从q处出发向上而行）。在某一点（令

其为r）处，边界必将与l再次相交。若$r \neq p$（如 图 3-14(a)所示），则在从q通往r的沿途上，我们所遇到的位置最高的那个顶点必然是一个分裂顶点，此时引理成立。

反之，若$r = p$（如 图 3-14(b)所示），则我们再一次沿着$\mathcal{P}$的边界，从q出发行进——不同的是，此次行进的方向与上次相反。与上面同理，我们将再次与l相交。令该交点为r'。我们断言，不可能有$r' = p$——否则，$\mathcal{P}$的边界与l只相交两次，这与“l和$\mathcal{P}$相交出多于一个连通子集”的前提相矛盾。因此就有$r' \neq p$，而这意味着，在从q通往r'的沿途中，所遇到的位置最低的那个顶点，必然是一个汇合顶点。 ❑

根据〖引理 3.4〗，只要将其中的分裂顶点和汇合顶点都消除掉，也就完成了将$\mathcal{P}$划分为多个y-单调块的任务。为此，需要在每个分裂顶点处增加一条向上的对角线，也要在每个汇合顶点处增加一条向下的对角线[①]。当然，这些对角线必须互不相交。一旦这些工作完成，$\mathcal{P}$也就已经被划分为多个y-单调块了。

首先来看看，在一个分裂顶点处应该如何引入一条对角线。这里采用平面扫描的方法。按照顺时针的方向，令$\mathcal{P}$的所有顶点排列为$v_1, v_2, \ldots, v_n$。再令$\mathcal{P}$的各边为$e_1, e_2, \cdots, e_n$，其中对任何的$1 \leq i < n$，都有$e_i = \overline{v_i v_{i+1}}$；另外，$e_n = \overline{v_n v_1}$。按照平面扫描算法，一条假想的水平扫描线l自上而下地扫过整个平面。在一些被称为事件点（event point）的位置，扫描线会稍做停留。就目前这一问题而言，这些事件点包括$\mathcal{P}$的所有顶点；不过，在整个扫描的过程中，不会产生任何新的事件点。所有的事件点被组织成一个事件队列（event queue）$\mathcal{Q}$。该事件队列实际上是一个优先队列，各顶点的优先级就是其各自的y-坐标[②]。如果两个顶点的y-坐标相同，则居于左边（x-坐标更小）的那个顶点具有更高的优先级。这样，每次只需$O(\log n)$时间，就可以找出下一待处理的顶点。（既然在扫描过程中不会出现新的事件，不妨在扫描之前将所有顶点按照y-坐标排一次序——经过这一预处理，每次只需$O(1)$时间就可以确定下一事件点。）

扫描的目的，是为了将每个分裂顶点，与位于其上方的某个顶点联接起来，从而引入一条对角线。如 图 3-15 所示，试考虑扫描线触及某个分裂顶点v_i的时刻。此时，应该将v_i与哪个顶点相联呢？与v_i相距较近的顶点，是一个不错的选择——这样，在将它与v_i联接起来之后，联线不与$\mathcal{P}$的任何边相交的可能性更大。让我们更准确地做一解释。沿着当前的扫描线，令居于v_i的左侧、与之相邻的那条边为e_j；令居于v_i的右侧、与之相邻的那条边为e_k。

[①] 意为：将分裂顶点（汇合顶点）与位于其上（下）方的另一个顶点相联接，形成一条对角线。——译者

[②] y-坐标越大，优先级越高。——译者

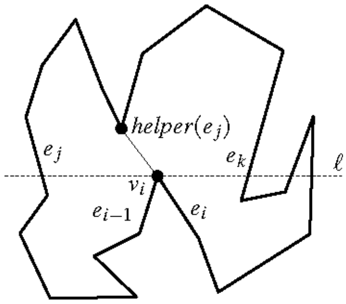


图3-15 分裂顶点的处理

现在考虑介于e_j和e_k之间、位于v_i上方的那些顶点，若这些顶点至少存在一个，则总可以将其中最低的那个与v_i联接起来（构成一条合法的对角线）。若这类顶点根本不存在，则可将v_i与e_j或e_k的上端点联接起来。无论如何，我们都将这个顶点称作"e_j的助手"（helper of e_j），记作helper(e_j)。按照正式的定义，helper(e_j)应该是"在位于扫描线上方、通过一条完全落在P内部的水平线段[①]与e_j相联的那些顶点中，高度最低的那个顶点"。请注意，helper(e_j)可能就是e_j自己的上端点。

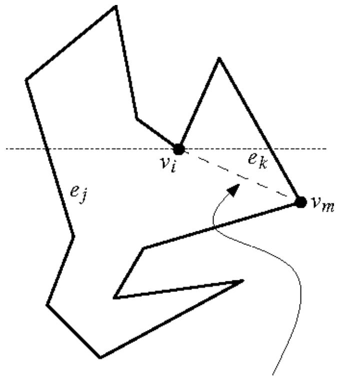


图3-16 汇合顶点的消除：当扫描线扫过v_m时，将输出一条对角线

这样，我们就知道了消除分裂顶点的方法——分别将它们与各自左侧那条相邻边的助手相联。那么，汇合顶点呢？从表面上看，它们似乎更难以消除——因为，对称地，它们各自需要借助一个位置更低的顶点，才能引入一条对角线。然而，位于扫描线下面的那些部分尚未访问到，所以在遇到一个汇合顶点时，并不能参照上面的方法构造出一条对角线。幸运的是，该问题并不像乍看起来那样困难。试考虑扫描线刚刚触及某一汇合顶点v_i的时刻。沿着扫描线的方向，令e_j（e_k）为居于v_i左（右）侧、与之相邻的边。请注意以下事实：在到达v_i的时候，它也就成为了e_j的新助手。这样，就可以从介于e_j和e_k之间、位于当前扫描线下方的所有顶点中，选出其中的最高者，然后将v_i与之相联。这个过程，与处理分裂顶点的情况正好相反——在那里，我们是在从介于e_j和e_k之间、位于当前

[①] 其长度可能为零。——译者

扫描线上方的所有顶点中，选出其中的最低者，然后将v_i与之相联。这也不值得奇怪——实际上，只要将上和下颠倒过来，汇合顶点也就相当于分裂顶点。当然，在扫描线触及v_i那一时刻，我们还不知道哪个才是位于扫描线下方的最高顶点。然而我们马上就会看到，这并不难判断出来。如图 3-16 所示，此后将遇到某个顶点v_m，它将取代v_i的地位，成为e_j的新助手——这时，v_m就是我们所寻找的顶点。因此，在每次更换某条边的助手时，都要通过检查以确认（被替换的）先前的助手是否为一个汇合顶点。如果是，就在新、老助手之间引入一条对角线。若新助手是一个分裂顶点，则这条对角线本来就需要被加入进来，以消除这一分裂顶点。若同时老助手是一个汇合顶点，则这条对角线将把一个分裂顶点和一个汇合顶点同时消除掉。还有一种可能：在扫描线越过v_i之后，e_j的助手不再会被更换——在这种情况下，可以将v_i与e_j的下端点联接起来。

按照上述方法，还需要找出居于每个顶点左侧、与之紧邻的那条边。为此，可使用一棵动态二分查找树$\mathcal{T}$，将$\mathcal{P}$中与当前扫描线相交的所有边存放在该树的叶子中。$\mathcal{T}$中所有叶子从左到右的次序，对应于这些边从左到右的次序。既然我们只关心在左侧与各分裂顶点或汇合顶点紧邻的边，故在$\mathcal{T}$中，只需存放$\mathcal{P}$的内部（在局部）位于其右侧的那些边[①]。对$\mathcal{T}$中的每一条边，我们都记录其对应的助手。树$\mathcal{T}$以及所存储的各边的助手，构成了扫描线算法的状态（Status）。随着扫描线的推进，状态会相应地变化：有些边可能开始与扫描线相交，原来与扫描线相交的一些边可能不再相交，同时某条边原先的助手可能会被新助手替换掉。

采用上述算法对 $\mathcal{P}$ 进行划分之后，得到的各个子多边形还必须经过后续的处理。为了能够方便地访问到这些子多边形，需要将由 $\mathcal{P}$ 导出的子区域划分（subdividsion）存储起来，并且将所有的对角线加入到双向链接边表 $\mathcal{D}$ 之中。我们假定，$\mathcal{P}$ 原本就是以双向链接边表（doubly-connected edge list）形式给出的；否则——比如，仅表示为所有顶点的一个逆时针列表——就需要首先为 $\mathcal{P}$ 构造出一个双向链接边表。随后，为每个分裂顶点和汇合顶点引入的对角线，都必须加入到这个双向链接边表之中。为了访问该双向链接边表，需要将状态结构与双向链接边表中对应的各边通过指针链接起来。借助于指针的操作，可以在常数时间内引入一条对角线。这样，就得到了如下的主算法：

算法 MAKEMONOTONE($\mathcal{P}$)

输入： 表示为双向链接边表 $\mathcal{D}$ 的一个简单多边形 $\mathcal{P}$

输出： $\mathcal{P}$ 的单调子多边形划分，同样地存储在 $\mathcal{D}$ 中

1. 以 y-坐标为优先级，将 $\mathcal{P}$ 的所有顶点组成一个优先队列 $\mathcal{Q}$
   若有多个顶点的 y-坐标相同，则 x-坐标小者优先级更高
2. 初始化一棵空的二分查找树 $\mathcal{T}$
3. **while** ($\mathcal{Q}$ 非空)

① 亦即所谓的“左边”。——译者

4.	**do** 从 Q 中取出优先级最高的顶点 v_i
5.	根据该顶点的类型，选用适当的子程序加以处理

接下来，详细介绍不同事件点的处理方法。刚开始阅读这些算法时，你可以暂不考虑任何退化情况；以后可以反过来验证，它们也能够正确处理各种退化情况。（当然，对在HANDLESPLITVERTEX第一行和HANDLEMERGEVERTEX第二行中出现的“在左侧紧邻”的概念，你必须给出恰当的定义。）在处理任何一个顶点的时候，我们都需要完成两项任务。首先，必须通过检查确定，是否需要引入一条对角线。若是分裂顶点，或者某条边的助手被替换了，而前任助手本身是一个汇合顶点，则需要引入对角线。其次，还要对状态结构T所存储的信息进行更新。处理各类事件的详细算法将在下面给出。你可以参照如 图 3-17 所示的例子来体会一下，在不同情况下将发生什么变化。

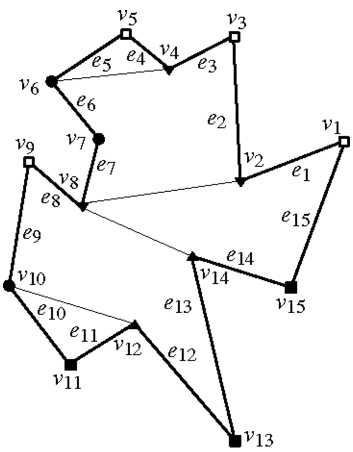


图3-17 单调剖分实例

算法	HANDLESTARTVERTEX(v_i)
1.	将 e_i 插入 T 中，将 helper(e_i)设为 v_i

例如，在如图实例中 v_5 处，要将 e_5 插入到树 T 之中。

算法	HANDLEENDVERTEX(v_i)
1.	**if** (helper(e_{i-1})为一个汇合顶点)
2.	**then** 在 v_i 和 helper(e_{i-1})之间生成一条对角线，并将该对角线插入到 D 中
3.	在 T 中删除 e_{i-1}

在上述运行实例中，当到达终止顶点 v_{15} 时，虽然 e_{14} 的助手为 v_{14}，但因为 v_{14} 不是一个汇合顶点，所以并不需要在此引入一条对角线。

算法	HANDLESPLITVERTEX(v_i)
1.	对 T 进行搜索，查找在左侧与 v_i 紧邻的那条边 e_j

2.	在 v_i 和 helper(e_j)之间生成一条对角线，并将该对角线插入到 $\mathcal{D}$ 中
3.	helper(e_j) ← v_i
4.	将 e_i 插入到 $\mathcal{T}$ 中，将 helper(e_i)设置为 v_i

对图例中的顶点v_{14}而言，在其左侧与之紧邻的边为e_9。该边的助手为v_8，故要在v_{14}与v_8之间引入一条对角线[①]。

算法	HANDLEMERGEVERTEX(v_i)
1.	**if** (helper(e_{i-1})为一个汇合顶点)
2.	**then** 在 v_i 和 helper(e_{i-1})之间生成一条对角线，并将该对角线插入到 $\mathcal{D}$ 中
3.	在 $\mathcal{T}$ 中删除 e_{i-1}
4.	对 $\mathcal{T}$ 进行搜索，查找在左侧与 v_i 紧邻的那条边 e_j
5.	**if** (helper(e_j)为一个汇合顶点)
6.	**then** 在 v_i 和 helper(e_j)之间生成一条对角线，并将该对角线插入到 $\mathcal{D}$ 中
7.	helper(e_j) ← v_i

在图例中的顶点 v_8 处，边 e_7 的助手为 v_2，它是一个汇合顶点，故需要在 v_8 与 v_2 之间引入一条对角线。

最后需要介绍的，只剩下处理普通顶点的子程序。对一个普通顶点的处理方法，取决于在其邻域 $\mathcal{P}$ 到底是处于它的左侧还是右侧。

算法	HANDLEREGULARVERTEX(v_i)
1.	**if** ($\mathcal{P}$ 的内部处于 v_i 的右侧)
2.	**then if** (helper(e_{i-1})是一个汇合顶点)
3.	**then** 生成一条对角线，联接 v_i 和 helper(e_{i-1})，并将该对角线插入到 $\mathcal{D}$ 中
4.	在 $\mathcal{T}$ 中删除 e_{i-1}
5.	将 e_i 插入到 $\mathcal{T}$ 中，将 helper(e_i)设置为 v_i
6.	**else** 对 $\mathcal{T}$ 进行搜索，查找在左侧与 v_i 紧邻的那条边 e_j
7.	**if** (helper(e_j)是一个汇合顶点)
8.	**then** 在 v_i 和 helper(e_j)之间生成一条对角线，并将该对角线插入到 $\mathcal{D}$ 中
9.	helper(e_j) ← v_i

比如在图例中的普通顶点 v_6 处，需要在 v_6 与 v_4 之间引入一条对角线。

[①] 还要将 e_{14} 插入到 $\mathcal{T}$ 中，并将 e_9 和 e_{14} 的助手都设置为 v_{14}。——译者

现在只需证明：算法 MAKEMONOTONE 的确能够正确地将 P 划分为多个单调块。

〖引理 3.5〗

通过引入一系列互不相交的对角线，算法 MAKEMONOTONE 能够将 P 划分为多个单调子多边形。

〖证明〗

不难看出：对P划分之后，每一子块都不再含有分裂顶点或汇合顶点。故由〖引理 3.4〗，每一子块都是单调的。于是只需证明：引入的每一条线段，都是合法的对角线（换而言之，它们不会与P的任何边相交）；此外，这些对角线之间也不会相交。为此，我们需要证明：每一条新引入的线段，都不会与P的任何边相交，也不会与此前引入的任何线段相交。我们只对由子程序 HANDLESPLITVERTEX 所引入的线段给出证明。至于由其它子程序（HANDLEENDVERTEX、HANDLEREGULARVERTEX以及HANDLEMERGEVERTEX）引入的线段，证明的过程都是类似的。我们还假定，各顶点的y-坐标互异——将这一结果推广至一般情况，是相当简单的。

试考察在到达 v_i 的高度时，由 HANDLESPLITVERTEX 所引入的对角线 $\overline{v_m v_i}$。令在 v_i 左侧与其紧邻的那条边为 e_j，而在 v_i 右侧与其近邻的那条边为 e_k。于是，在触及 v_i 时，helper(e_j) = v_m。

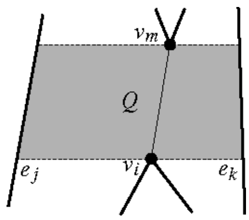


图3-18 介于v_m和v_i之间水平梯形Q内部必空

首先说明：$\overline{v_m v_i}$不会与P的任何一条边相交。为此，可考察 图 3-18 中由e_j、e_k以及分别通过v_m和v_i的两条水平线所确定的那个四边形Q。我们断言：Q的内部不含P的任何顶点。否则，v_m就不可能成为e_j的助手。现在假设：$\overline{v_m v_i}$与P的某条边相交。既然这条边的端点都不可能落在Q中，而且多边形的边互不相交，故这条边要么跨越联接于v_m与e_j之间的水平线段，要么跨越联接于v_i与e_j之间的水平线段。然而，这两种情况都不可能出现——因为，无论是对v_m还是v_i而言，在其左侧与之紧邻的边都是e_j。因此，$\overline{v_m v_i}$不会与P的任何边相交。

最后，再来考虑此前所引入的那些对角线。既然 Q 的内部不含 P 的任何顶点，而且此前所引入的每一条对角线的两个端点都要高于 v_i，故它们都不可能与$\overline{v_m v_i}$相交。 ❑

下面对该算法的运行时间做一分析。构造优先队列*Q*需要线性的时间 [①]，而树*T*的初始化只需常数时间。在扫描过程中，每次处理一个事件点，都只需要对*Q*执行一次操作；对于树*T*，最多只分别做一次查找、一次插入和一次删除；对于*D*，最多插入两条对角线。无论是优先队列，还是平衡查找树，都可以在*O*(**log**n)时间内完成一次查找或一次更新；而将一条对角线插入到*D*中，只需*O*(1)时间。因此，只需*O*(**log**n)时间，就可以处理完一次事件，于是整个算法所需的时间就是*O*(n**log**n)。显然，该算法只需线性的空间——在*Q*中，每个顶点至多被存储一次；在*T*中，每条边至多存储一次。这样，结合〖引理 3.5〗，就可以得出如下定理：

〖定理 3.6〗

使用 *O*(n)的存储空间，可以在 *O*(nlog**n)时间内将包含 n 个顶点的任何简单多边形分解为多个 y-单调的子块。**

3.3 单调多边形的三角剖分

在上面我们已经看到，如何在 *O*(n**log**n)时间内，将任一简单多边形划分成多个 y-单调的子块。这个结果本身并没有什么价值。本节将说明：可以在线性的时间内，完成对单调多边形的三角剖分。只有将这一结果与前一节的结果联系起来，才能得出结论：对任何简单多边形的三角剖分，都可以在 *O*(n**log**n)时间内完成——前一节的开头曾描述过一个需要平方量级时间的算法，现在的这个结果无疑是一个很大的改进。

给定一个包含 n 个顶点的 y-单调多边形 *P*。暂且假定，*P* 是严格 y-单调的（strictly y-monotone）——也就是说，它不仅是 y-单调的，而且不含任何水平边。这样，在从最高的顶点出发，沿着 *P* 的左边界或右边界走向最低顶点的沿途，我们的高度一直都在下降。正是由于这一性质，单调多边形的三角剖分才变得很容易——可以从最高顶点开始，同时沿着 *P* 的（左、右）两条边界链，走向最低的顶点；在此过程中，只要有可能，就引入对角线。接下来，就详细介绍三角剖分这一的贪婪算法（greedy algorithm）。

该算法按照y-坐标递减的次序，依次处理各个顶点。若有两个顶点的y-坐标相等，则其中靠左的顶点将被优先处理。该算法需要利用一个栈*S*做为辅助的数据结构。一开始，该栈为空；在算法过程中，它存放了在*P*中已经被发现、却仍然可以生出更多对角线的那些顶点。在处理每个顶点的时候，我们将尽可能地在这个顶点与栈中的各顶点之间引入对角线。这些对角线会从*P*中分离出若干三角

[①] 比如，采用 Robert-Floyd 算法。——译者

形。已经做过一些处理，但尚未从原多边形中分离出来的那些顶点（亦即仍滞留在栈中的顶点）都散落在$\mathcal{P}$中尚未被三角剖分的部分（与已处理过的部分之间）的边界上。这些顶点中位置最低的那个（亦即最后开始接受处理的那个顶点），就位于栈顶的位置；高度次低的那个顶点，则位于次栈顶的位置；依此类推。如 图 3-19 所示，在已经被发现的那些顶点之上，$\mathcal{P}$中还有一些部分尚待剖分，这些部分具有特定的形状——犹如一个倒置的漏斗。这个漏斗（左或右）一侧的边界，由$\mathcal{P}$的某条边独立地界定；而沿着它在另一侧的边界，所有的顶点都是凹顶点（reflex vertex）——亦即，这些顶点各自对应的内角都不小于 180º（其中最高的那个顶点除外，它是凸的）。在我们处理完接下来的一个顶点之后，这个性质依然保持——也就是说，这是该算法所具有的一个不变性。

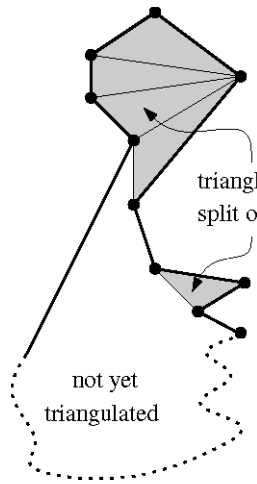


图3-19 已经三角剖分的部分与尚未三角剖分的部分

现在来看看，在处理下一个顶点的时候，可以引入哪些对角线。分两种情况处理：接受处理的下一顶点v_j，与栈中的那些凹顶点处于（漏斗的）同一侧；或者，处于对面的另一侧。若是后一种情况（图 3-20），则v_j必然就是独自界定该漏斗一侧边界的那条边e的下端点。鉴于其漏斗的形状，我们可以从v_j出发，与当前栈中除最后一个（即居于栈底的那个）顶点之外的每个顶点，分别联接一条对角线。而实际上，此时栈中的最后一个顶点，就是e的上端点——也就是说，该顶点实际上已经与v_j联接了。所有这些顶点都将从栈中弹出。此后，在v_j之上，原多边形中尚待三角剖分的部分，将由此前生成的、联接v_j与栈顶顶点的那条对角线界定；而该部分的范围，将在该顶点处向下方延伸——因此，这部分仍然是（倒立的）漏斗状，故上述不变性依然保持。该顶点以及v_j仍然属于多边形中尚待三角剖分的那部分，因此，需要将它们（再次）压入栈中。

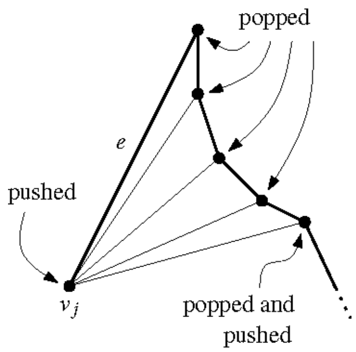


图3-20 接受处理的下一顶点v_j处于对面的另一侧

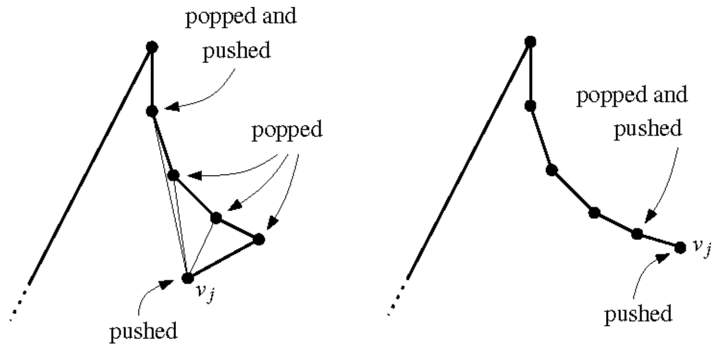


图3-21 当下一顶点与栈中各凹顶点处于（漏斗的）同一侧时，可能出现的两种情况

在另一种情况中，v_j与栈中的那些凹顶点同属于漏斗的一侧。此时，从v_j出发，就不见得能够与栈中的每一个顶点都联接一条（合法的）对角线。尽管如此，v_j还是能够与其中的某些顶点联接——这些顶点必然是依次相邻的，而且在栈中都位于顶部。因此可按如下方法处理：首先，从栈中弹出一个顶点（该顶点已经通过$\mathcal{P}$的一条边，与v_j联接）；然后，依次从栈中弹出各顶点，并将其与v_j联接。不断重复这一过程，直到不能如此联接的某个顶点。为确定能否在v_j与栈中的某个顶点v_k之间联接一条对角线，只需检查v_j、v_k以及此前刚刚被弹出的那个顶点[①]。一旦遇到一个不能与v_j相联的顶点，就将被弹出的前一顶点重新压入栈中。若此前确实联接出过至少一条对角线，则该顶点就是最后那条对角线的端点[②]；若根本就没有生成过任何对角线，则沿着$\mathcal{P}$的边界该顶点必然与v_j相邻（参见 图 3-21）。完成上述操作之后，将v_j再次压入栈中。此时，无论是哪种情况，不变性又重新恢复了——漏斗的一侧边界由多边形的一条边独立界定，而另一侧边界则由一串（依次相邻的）凹顶点确定。由此可以得出如下算法（实际上，该算法与第 1 章中的凸包算法很相似）：

算法 TRIANGULATEMONOTONEPOLYGON($\mathcal{P}$)
输入：表示为双向链接边表 $\mathcal{D}$ 的一个严格 y-单调的多边形 $\mathcal{P}$
输出：$\mathcal{P}$ 的三角剖分，同样存储在 $\mathcal{D}$ 中

① 若此前最后弹出的顶点为 v_t，则只需检查在 v_t 处，v_j、v_t 和 v_k 是否定义了一个“左拐”（left-turn）。——译者

② 另一个端点是 v_j。——译者

```
1.   将 P 左、右侧边界上的所有顶点合并起来，按照 y-坐标排成一个递减的序列
     若有多个顶点的 y-坐标相等，则 x-坐标小者在前
     (* 令排序后的序列为 u1, …, un *)
2.   初始化一个空栈 S，然后将 u1 和 u2 压入其中
3.   for (j ← 3 to n-1)
4.      do if (uj 处于与 S 栈顶顶点对面的一侧)
5.            then 弹出 S 中的所有顶点
6.                 对于弹出的（除最后一个外的）每个顶点
                      在 uj 与该顶点之间生成一条对角线
7.                 将 uj-1 和 uj 压入 S
8.            else 弹出 S 的栈顶
9.                 不断检查当前栈顶处的顶点：
                      只要它与 uj 的联线完全落在 P 的内部，就弹出该顶点
                   把这些联线当作对角线，插入到 D 中
                   将最后弹出的那个顶点，重新压入 S 中
10.                将 uj 压入 S 中
11.  将 un 与栈中（除第一个和最后一个外的）每个顶点相联构成对角线
```

这个算法需要运行多长的时间呢？第 1 步需要线性的时间，第 2 步需要常数时间。for-循环共有 n-3 轮，每一轮最多可能需要线性的时间。然而，在每一轮**for**-循环中，需要压入*S*的顶点不会超过两个。因此，加上第 2 步中的两次压栈操作，压栈操作的总数不会超过 2n – 4[①]。自然地，退栈操作的次数不可能超过压栈，故**for**-循环总共的运行时间为*O*(n)。算法最后一步所需的时间，也不会超过线性的量级。总而言之，该算法的运行时间为*O*(n)。

〖定理 3.7〗

由 n 个顶点组成的任一严格 y-单调多边形，都可以在线性时间内被三角剖分。

我们希望把单调多边形的三角剖分算法，做为对任意简单多边形进行三角剖分的一个子程序。按照这一构思，首先要将多边形划分为若干单调子块，然后分别对各单调子块进行三角剖分。看起来，似乎所有必需的条件都已具备。然而，还有一个问题——本节一直假定：输入都是严格 y-单调的多边形；然而按照前一节所介绍的算法，生成的单调子块中有可能含有水平边。你应该记得，在前一节中，对于 y-坐标相等的顶点，我们是按照自左向右的次序进行处理的。其效果等同于沿顺时

① 这被称为分摊分析（amortized analysis）。独立地看，每一轮循环最多都可能进行线性次栈操作；然而总体统计，所有循环总共需要进行的栈操作也不过是线性次。——译者

针方向，将整个平面做一足够小角度的旋转，从而使得任何两个顶点都不会处于同一水平高度上。于是，在这个经过小角度旋转之后的平面上，由上节的算法划分出来的单调子多边形，必然都是严格单调的。这样，只要我们依然按照自左向右的次序处理那些 y-坐标相同的顶点（这等同于在旋转后的平面上进行计算），本节的三角剖分算法就能正常地工作。因此，我们可以将这两个算法结合起来，得出一个适用于任何简单多边形的三角剖分算法。

这个三角剖分算法的运行时间有多长？由〖定理 3.6〗，可在O(n**log**n)时间内将一个多边形分解为多个单调子块。第二个阶段可采用本节的算法，在线性时间内对各单调子块分别进行三角剖分。由于所有子块包含的顶点总数为O(n)，故第二个阶段总共需要O(n)时间。由此可以归纳出如下结论：

〖定理 3.8〗

使用 O(n)的存储空间，可以在 O(n**log**n)时间内对由 n 个顶点组成的任一简单多边形进行三角剖分。

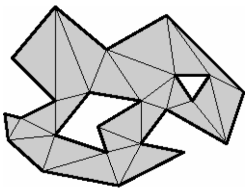


图3-22 带洞多边形的三角剖分

我们已经知道了应该如何对简单多边形进行三角剖分。但是，对那些内部含有空洞的多边形（图 3-22）呢？它们也能够如此轻易地被三角剖分吗？答案是肯定的。实际上，我们所介绍的算法同样适用于内部存在空洞的多边形——在将一个多边形分解为多个单调子块的过程中，我们本来就没有要求多边形是简单的。该算法甚至还适用于另外一种更具一般性的情况——给定一个平面子区域划分S，要求对S进行三角剖分。对这一问题更准确的描述是：如果S的所有边都落在某一包围框（bounding box）B的内部，我们希望构造出由互不相交的对角线——也就是联接于S和B的顶点之间、与S的边不相交的线段——组成的一个极大集合，这些对角线将B划分为多个三角形。

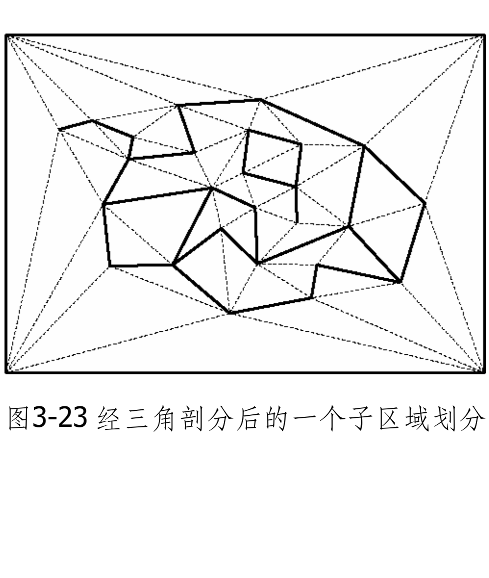


图3-23 经三角剖分后的一个子区域划分

图 3-23 所显示的，就是一个子区域划分的三角剖分。图中，用粗线条来表示原子区域划分的边以及包围框的边。可以采用本章所介绍的算法，来构造这样一个三角剖分——首先，将该子区域划分分解为多个单调子块；然后，分别对各子块做三角剖分。由此可以得出如下定理：

〖定理 3.9〗

使用 O(n)存储空间，可以在 O(n**log**n)时间内对包含 n 个顶点的任一平面子区域划分进行三角剖分。

3.4 注释及评论

艺术画廊问题是由Klee在 1973 年与Vasek Chvatal的一次交谈中提出的。1975 年，Chvatal[128]第一次证明：$\lfloor \frac{n}{3} \rfloor$台摄像机总是足够的，而且有时是必需的。这一结论被称为艺术画廊定理，或者看守者定理（watchman Theorem）。Chvatal的证明十分繁琐。本章引述的证明更加简明，它是由Fisk[178]给出的。该证明建立在Meisters的双耳定理（Two Ears theorem）之上，由此可以轻松地导出“任意简单多边形经三角剖分后，所对应的图均可 3-染色”这一结论。“求任一简单多边形所需看守的最少数目”这一算法问题，已分别由Aggarwal[10]以及Lee和Lin[246]证明是NP-难的。O'Rourke的专著[298]以及Shermer的综述 [355]，都对艺术画廊问题及其形形色色的变种做了详尽的讨论。

将一个多边形（或者其它类型的区域）分解为简单子块的策略，在很多的问题中都十分有用。这种简单的子块往往就是三角形——此时，相应的分解结果称作一个三角剖分。然而在另一些时候，也可能采用其它的形状，比如四边形（quadrilateral）或者梯形（trapezoid）——参见第 6、9 和 14 章。在此仅讨论与多边形三角剖分有关的结果。本章所介绍的单调多边形三角剖分的线性时间算法，是由Garey等人提出的 [188]；而将多边形分解为单调子块的平面扫描算法，则是由Lee和Preparata[250]提出的。Avis和Toussaint[32]以及Chazelle[85]也分别给出了可以在O(n**log**n)时间内对简单多边形进行三角剖分的不同算法。

能否在o(n**log**n)时间 [①]内对简单多边形进行三角剖分？在计算几何（computational geometry）界，这个问题许久都没有答案。（对于内部存在空洞的子区域划分，Ω(n**log**n)的确是一个下界。）在本章中我们已经看到，对于单调多边形而言，的确可以如此。此外，对于很多其它类型的多边形，都找到了线性时间的三角剖分算法 [108][109][170][184][214]；然而，就一般的简单多边形而言，这个

[①] 用大 O 记号（Big-O Notation）表示的复杂度，可能是渐进紧的（asmpototically tight），也可能不是。比如 2n**log**n ∈ O(n**log**n)和 2n ∈ O(n**log**n)都成立，但前者是紧的，而后者却不是。为了加以区分，可以通过小 o 记号（small-O notation）来表示非紧的复杂度上界。比如，2n ∈ o(n**log**n)，但 2n**log**n ∉ o(n**log**n)。因此，原文此处所提的问题是：能否在严格少于 n**log**n 的时间（比如 n**loglog**n 或线性的时间）内，对简单多边形进行三角剖分。——译者

问题在许多年内一直悬而未决。1988 年，Tarjan和Van Wyk[368]突破了$O(n\mathbf{log}n)$的限制，他们提出了一个$O(n\mathbf{loglog}n)$的算法。后来，Kirkpatrick等人 [237]对他们的算法进行了简化。在设计更快速的（三角剖分）算法时，随机化（randomization）技术——将在本书的第 4、6 以及 9 章中用到——被证明是一种有力的手段。借助于这一技术，Clarkson等人 [134]、Devillers[141]以及Seidel[345]都陆续提出了各自的$O(n\mathbf{log}^*n)$运行时间的算法（这里的$\mathbf{log}^*n$是反复地对n取对数，直到结果小于 1 之前所需取对数的次数）。这些算法不仅比$O(n\mathbf{loglog}n)$的算法稍快一些，而且更为简单。第 6 章将介绍一个算法，构造平面子区域划分的梯形分解（trapezoidal decomposition），Seidel所提出的算法与这一算法密切相关。然而，“能否在线性时间内完成简单多边形的三角剖分”这一问题，依然没有答案。1990 年，这一个问题终于被Chazelle[92][94]圆满解决，他给出了一个线性时间的确定性算法（尽管十分晦涩难懂）。

在三维空间中，与多边形三角剖分相对应的问题可以表述如下：给定一个多胞体（polytope），要求将它分解为互不相交的四面体（tetrahedron），其中各四面体的所有顶点，都必须是原多胞体的顶点。多胞体的这种分解被称为四面体剖分（tetrahedralization）。与其对应的二维问题相比，这一问题要难得多。事实上，若只允许使用原多胞体的顶点，则对于有些多胞体来说，根本就不存在这种剖分方案。Chazelle[86]证明：（对于任何足够大的n，都）存在某个包含n个顶点的简单多胞体（simple polytope），需要用到$\theta(n^2)$个额外的顶点[①]（才能对其做四面体剖分）；反过来，只要允许使用这样多个额外顶点，对任一简单多胞体我们都必然能够做四面体剖分。后来，Chazelle与Palios[110]一道将这一界限改进到$\theta(n + r^2)$，其中r为多胞体中包含的凹边（reflex edge）数目。计算这种剖分的算法，运行时间为$O(nr + r^2\mathbf{log}r)$。“判断任一给定简单多胞体，能否在不引入额外顶点的情况下被四面体剖分”这一问题，是NP-完全的（NP-complete）[330]。

3.5 习题

习题 3.1 试证明：任何（即使是有空洞的）多边形都存在一个三角剖分。关于其三角剖分中所含三角形的数目，你能给出什么结论?

习题 3.2 在一个所谓的“矩形多边形”（rectilinear polygon）中，所有边的方向不是水平的就是垂直的。考察包含 n 个顶点的矩形多边形 P。试举例说明：为了看守这种多边形，有时至少需要$\lfloor \frac{n}{4} \rfloor$台摄像机。

习题 3.3 试证明或证伪：单调多边形的三角剖分的对偶图，必然是一条链（chain）——亦即，

[①] 这类起辅助作用的点，称作 Steiner 点（Steiner point）。——译者

该图中每个顶点的度数均不超过 2[①]。

习题 3.4 给定由任意n个顶点组成的一个简单多边形P，已经通过一组对角线将其分解为多个凸四边形。此时，只需多少台摄像机就足以看守P？这一结论为什么不会与艺术画廊定理相矛盾？[②]

习题 3.5 试以伪代码的形式给出一个算法，对经过三角剖分之后的任一简单多边形进行 3-染色。该算法的时间复杂度必须是线性的。

习题 3.6 试给出一个算法，在 $O(n\log n)$时间内，在一个简单多边形中找到一条对角线，这条对角线不仅将该多边形划分为两个子简单多边形，而且每个子多边形所包含的顶点数目均不超过$\lfloor \frac{2n}{3} \rfloor + 2$。提示：利用多边形三角剖分的对偶图。

习题 3.7 给定由任意 n 个顶点组成的一个简单多边形 P，而且已经将它分解为多个单调子块。试证明：所有子块中所含顶点数的总和仍然是 $O(n)$。

习题 3.8 本章所介绍的将简单多边形分解为单调子块的算法，需要为经过分解后的多边形建立一个双向链接边表结构。在算法运行过程中，不断会有新的边（具体讲，就是为了消除分裂顶点和汇合顶点而引入的那些对角线）加入到 DCEL 中。一般而言，在 DCEL 结构中加入一条边，并不能够在常数时间内完成。试分析，加入一条边为什么可能要花费超过常数的时间。然后再说明，尽管如此，我们的多边形分解算法依然能够在$O(1)$时间内加入一条新边。

习题 3.9 试证明：若多边形只含 $O(1)$个拐点，则可以使本章所介绍算法的时间复杂度降至 $O(n)$。

习题 3.10 能否应用本章所介绍的算法，对含有 n 个点的任一点集进行三角剖分？如果可以，试说明应该如何使这一三角剖分算法更为有效。

习题 3.11 试给出一个有效的算法，判断包含n个顶点的任一多边形P是否关于（不一定是水平的或垂直的）某条直线单调。[③]

[①] 对“链”更准确的定义是：只有两个节点的度数为 1，所有其它节点的度数均为 2。——译者

[②] 可参考[298]第二章。——译者

[③] 参见：F. P. Preparata and K. Supowit, Testing a Simple Polygon for Monotonicity. Information Processing Letters 12, 161-164 (1981)。——译者

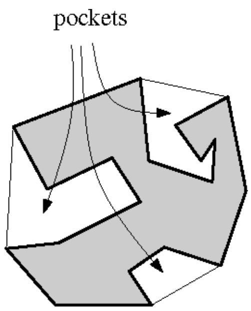


图3-24 简单多边形的口袋

习题 3.12 所谓简单多边形的“口袋”（pocket），指的是虽然在多边形之外，却在多边形凸包之内的那些区域。设 P_1 为由 m 个顶点组成的一个简单多边形，而且不仅给出了 P_1 的三角剖分，还给出了其中所有口袋的信息。设 P_2 为由 n 个顶点组成的另一个凸多边形。试说明：可以在 $O(m + n)$时间内，计算出两个多边形的交 $P_1 \cap P_2$。

习题 3.13 给定多边形 P 的一个三角剖分。完全落在 P 内部的任一线段，与其中各对角线之间可能的最大交点数目，被称为 P 的这一三角剖分的“穿刺数”（stabbing number）。试给出一个算法，构造任一凸多边形的三角剖分，并保证所生成三角剖分的穿刺数不超过 $O(\mathbf{log}n)$。

习题 3.14 给定由 n 个顶点组成的任一简单多边形 P，以及其内部的一点 p。试说明，如何在 P 的内部计算出与 p 可见的区域。

4

线性规划：铸模制造

今天我们所能看到的大多数物体——从轿车车厢到塑料杯和餐具——都是通过某种自动制造工艺制造出来的。在这个过程中，无论是设计阶段还是实际的制造阶段，计算机都扮演了一个重要的角色；而对于现代的工厂来说，CAD/CAM 工具已经成为了其中至关重要的一个组成部分。具体应该采用什么方法来制造某种物体，取决于一些因素，比如用来制造该物体的材料、物体的外形以及是否需要大规模生产等。在生产塑料或金属物体的时候，一种常用的方法就是利用铸模（mold），本章将研究这一过程中的一些几何问题。对于金属物体而言，这一过程也常常被称作铸造（casting）。

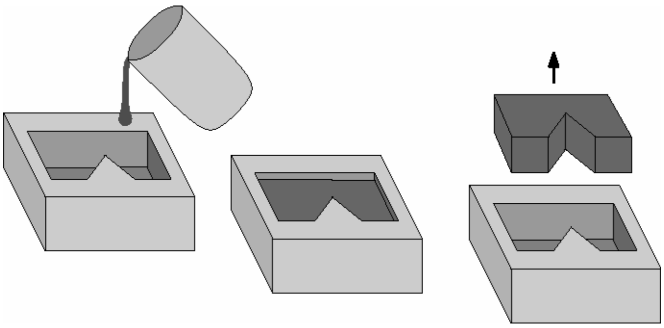


图4-1 铸造的过程

铸造过程如 图 4-1 所示：液态金属被倒入铸模中，待凝固之后再将成形的物体从中抽取出来。然而，最后一步并不总是想乍看起来那么轻而易举——有时，成形后的物体可能会被卡在铸模当中，为把物体抽取出来，将不得不把铸模打破。为解决这一问题，有时需要尝试使用别的铸模。然而对某些物体而言，根本就设计不出任何好的铸模——球体就是这样的一个例子。这正是本章将要探讨的问题——给定一个物体，能否设计出一种铸模，使成形后的物体能够顺利地从中抽取出来？

我们将做如下限定。首先，假设需要制造的物体都是多面体形状的。其次，我们只考虑连通为一体的铸模，而排除掉由有两块或更多块组成的铸模。（若铸模可以由多块组成，则有可能制做出诸如球体之类的物体；反之，若要求铸模本身必须是完整的一块，这类物体就不可能铸造出来。）我们最后还要求，只通过一次平移运动，即可将物体从铸模中抽取出来。幸运的是，对于许多物体来说，平移运动已经足够了。

4.1 铸造中的几何

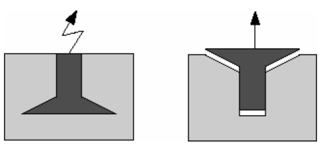


图4-2 朝向的不同选择

给定某一物体，能否通过某种铸造工艺制造出该物体呢？为回答这类问题，必须设计出一种适当的铸模。铸模内部空洞的形状，取决于物体的外形；即使是同一物体，按照摆放方向的不同，也将对应于不同的铸模。问题的关键在于摆放朝向的选择——如 图 4-2 所示，沿某些方向摆放物体，物体成形后将无法从对应的铸模中抽取出来；反之，若沿着另外一些方向放置，则可顺利地抽取出来。关于物体摆放的朝向，一个显而易见的必要条件是：物体必须具有一张水平的顶面（top facet）；另外，物体与铸模不相接触的面仅限于这一张。因此，物体有多少张面，相应地就有多少种可能的摆放朝向——相应地，也就有多少种可能的铸模。给定一个物体，在对应于这些方向的各个铸模中，若至少有一个可以使成形后的物体从中顺利取出，则称之为可铸造的（castable）。下面将着重讨论

的问题是：给定一个铸模，如何确定成形于其中的物体能否通过一次平移运动从中抽取出来。这样，为了确定物体的可铸造性（castability），我们只要对所有可能的方向逐一尝试。

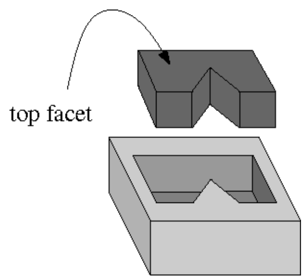


图4-3 多面体的顶面

设𝒫为一个三维的多面体——亦即由多张二维小平面（facet）围成的一个三维形体——且其中有一个小平面被指定为顶面（图 4-3）。（关于多面体，我们不打算给出准确而形式化的定义。这类定义过于复杂且没有必要。）我们假定，铸模的外轮廓呈立方体形状，而其内部的空洞则与𝒫完全一致。如果将这个多面体放入到铸模中，其顶面将与铸模的顶面共面平齐——我们假设这张平面与xy-平面平行。也就是说，若试图将𝒫抽取出来，则在铸模的上方不会受到任何不必要部分的阻挡。

除了顶面之外，𝒫 的其它小平面都被称作普通面（ordinary facet）。𝒫 的任何一张普通面 f，都与铸模的某张小平面相对应，我们记之为$\hat{f}$。

只做一次平移运动，能否将 𝒫 从铸模中抽取出来？我们希望对此做出判断。换而言之，我们所要判断的是：是否存在某一方向$\vec{d}$，使得在沿该方向将 𝒫 平移至无穷远的过程中，𝒫 与铸模体的任何部分都不相交。需要注意的是，这里允许 𝒫 紧贴铸模的边界运动。既然在 𝒫 的各张小平面中，只有顶面不与铸模接触，故移出的方向必然朝上——也就是说，其在 z-方向上的分量必须为正。当然，这只是移出方向应具备的必要条件之一；关于合法的移出方向，还需要确定更多的限制条件。

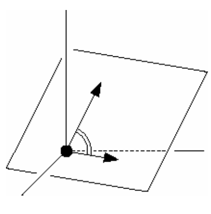


图4-4 空间中向量的夹角

任取𝒫的一张普通面f。相对于铸模上与之对应的小平面$\hat{f}$，f可能的移出方式不外乎两种：要么与$\hat{f}$逐渐远离，要么紧贴$\hat{f}$滑动。为了更准确地表述这一限制条件，我们需要对三维空间中任意两个向量所构成的角度作出定义。我们的定义如下。如 图 4-4 所示，考虑由这两个向量所生成的那张平

面（这里假定两个向量都起自于坐标原点）；在该平面上，这两个向量可以确定出两个角度（其和为 2π），其中较小的那个，被定义为这两个向量所成的角度。现在，若将f的外法矢（outward normal）记作$\vec{\eta}(f)$，则沿着与$\vec{\eta}(f)$的夹角小于 90^o的任何方向的平移运动，都会受到$\hat{f}$的阻挡。因此，$\vec{d}$所应该具备的一个必要条件就是，相对于$\mathcal{P}$的任何一张普通面的外法矢，$\vec{d}$与之所成的角度都必须至少是 90°。反过来，以下引理指出，这一条件也是充分的。

〖引理 4.1〗

沿着某个方向$\vec{d}$通过一次平移，多面体 $\mathcal{P}$ 能够从其铸模中抽取出来，当且仅当相对于 $\mathcal{P}$ 的每一张普通面的外法矢，$\vec{d}$与之所成的角度都至少为 90°。

〖证明〗

“仅当”的方向很容易：无论$\vec{d}$与哪一条外法矢$\vec{\eta}(f)$所成的角度小于 90°，则在沿着$\vec{d}$方向试图进行平移运动时，f 内部的任何一个点 q 都将与铸模发生碰撞。

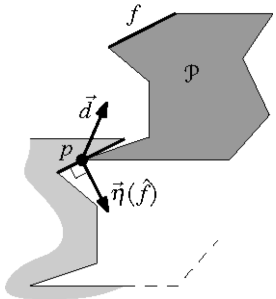


图4-5 点p与铸模小平面$\hat{f}$发生碰撞的时刻

为了证明“当”的方向，我们反过来假设，在沿着$\vec{d}$方向进行平移过程中的某一时刻，$\mathcal{P}$与铸模发生碰撞。我们可以证明：必然存在某条外法矢，它与$\vec{d}$所成的角度小于 90°。为此，我们进一步假设：$\mathcal{P}$上的一个点p与铸模的某张小平面$\hat{f}$发生碰撞。这也就是说，p正试图进入铸模的内部。于是如 图 4-5 所示，$\hat{f}$的外法矢$\vec{\eta}(\hat{f})$必然与$\vec{d}$形成一个大于 90^o的夹角。现考察$\mathcal{P}$上与$\hat{f}$对应的那张普通面f。此时，$\vec{d}$与f的外法矢所成的夹角必然小于 90°。 ❑

根据〖引理 4.1〗，可以得出一个有趣的推论：若可以通过多次小幅度的平移将$\mathcal{P}$抽取出来，则必然可以仅通过一次平移就将它抽取出来。因此，对于将（铸造的）物体从铸模中抽取出来这一任务而言，纵然允许进行多次平移，也不会有任何帮助。

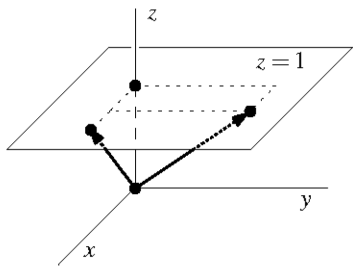


图4-6 z-分量为正的每一个方向，都对应于平面z = 1上的一个点

这样，我们的任务只不过是要找出一个特定的方向$\vec{d}$，使之相对于P上任何一张普通面的外法矢所成的夹角至少为 90°。三维空间中的任何一个方向，都可以表示为起始于原点的某一向量。前面的分析告诉我们：只需将注意力放在z-分量为正的那些方向上。如 图 4-6 所示，每一个这样的方向，都可以表示为平面z = 1 上的一个点——也就是说，点(x, y, 1)表示与向量(x, y, 1)所对应的那个方向。这样，平面z = 1 上的每一个点，都唯一地确定了一个方向；反过来，每一个z-分量为正的方向，都可以由该平面上的某个点唯一确定。

〖引理 4.1〗给出了合法移出方向$\vec{d}$的充要条件。那么，应如何将这些条件转换到我们用以表示方向的这张平面上呢？任取一张普通面的外法矢$\vec{\eta} = (\vec{\eta}_x, \vec{\eta}_y, \vec{\eta}_z)$。方向$\vec{d} = (d_x, d_y, 1)$与$\vec{\eta}$所成夹角不小于 90°，当且仅当$\vec{d}$与$\vec{\eta}$的点积非正。这样，由每一张普通面都可导出如下形式的一个限制条件：

$$\vec{\eta}_x d_x + \vec{\eta}_y d_y + \vec{\eta}_z \leq 0$$

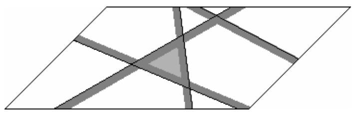


图4-7 可行的方向，对应于一组半平面的公共交集

在平面z = 1 上，这样一个不等式所描述的正好是一张半平面——也就是在平面上位于某条直线左（或右）侧的部分[①]。（当然，对于水平的小平面，由于$\vec{\eta}_x = \vec{\eta}_y = 0$，故这句话不成立。在这种情况下，对应的限制条件要么绝不可能满足，要么总是满足——具体是哪种情况，很容易检测出来。）于是如 图 4-7 所示，P上每一张非水平小平面，都在平面z = 1 上定义了一张闭的半平面；若所有这些半平面的公共交集非空，则其中的任何一个点，都对应于一个可以将P顺利抽取出来的方向。当然，若这些半平面的公共交集为空，则意味着不可能将P从铸模中抽取出来。

如此，上述制造问题即转化为一个纯粹的平面几何问题：给定一组半平面，若其公共交集非空，

[①] 此处不等式含等号，故对应的半平面是闭集。亦即，除直线左（或右）侧的部分还包含直线上各点。——译者

则从中找出一点；否则，应能判断为空。若待制造多面体有n张小平面，则其对应的问题最多要考虑 n-1 张半平面（由顶面导出的那张不必考虑）。以下各节将说明，如上定义的平面几何问题可在线性的期望时间（expected time）内解决——第 4.4 节也对“期望”一词的含义做了明确定义。

不要忘了，该几何问题的原问题是：判断物体 $\mathcal{P}$ 能否从某个给定的铸模中抽取出来。虽然某一次判断的结果可能是否定的，但是对于同一个物体，由于其选取顶面的不同，还会有多种其它的铸模，使得 $\mathcal{P}$ 可以从其中的某些之中抽取出来。总之，为了判断某个物体 $\mathcal{P}$ 是不是可铸造的，只需分别将它的各张小平面当作顶面，逐一进行尝试。由此可以得出如下结论：

〖定理 4.2〗

任给由 n 张小平面围成的多面体 $\mathcal{P}$。使用 $O(n)$空间，可在 $O(n^2)$时间内判断 $\mathcal{P}$ 是否可铸造的。果真如此，还可在同样长的时间内，计算出一个可行的铸模，以及将 $\mathcal{P}$ 从中抽取出来的具体方向。

4.2 半平面求交

任给双变量线性约束条件（linear constraint）集 $H = \{h_1, h_2, \cdots h_n\}$，其中约束条件形式如下：

$$a_i x + b_i y \le c_i$$

其中 a_i、b_i 和 c_i 都是常数，且 a_i 与 b_i 不同为零。从几何角度看，每个约束条件都可以被理解为 $\mathbb{R}^2$ 空间中的一张闭的半平面，其边界为直线 $a_i x + b_i y = c_i$。本节所要讨论的问题是：找出同时满足全部 n 个约束条件的所有点$(x, y) \in \mathbb{R}^2$。亦即，我们希望找出落在 H 中所有半平面公共交集内部的所有点。（前一节已将铸造问题归结为“在一组半平面的公共交集中找出某个点”。相对于那个问题，此问题更具一般性。）

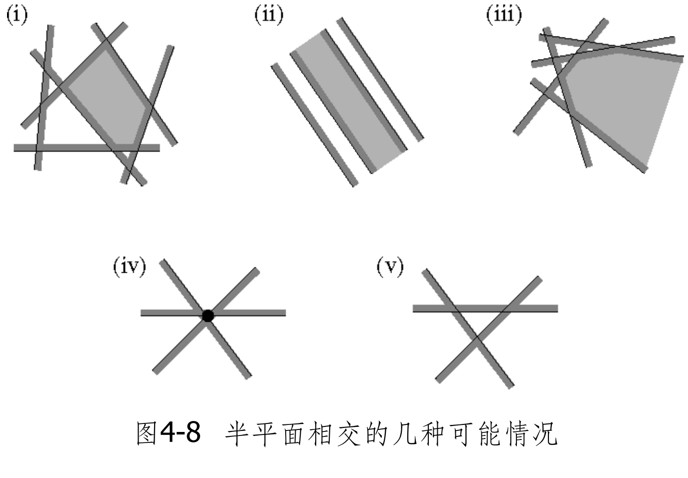


图4-8 半平面相交的几种可能情况

一组半平面公共交集的外形，不难确定——既然每张半平面都是凸的，且多个凸集的交依然是

凸的，故任何一组半平面的交仍然是平面上的一个凸集。位于该公共交集（若非空）边界上的任何点，必然来自某张半平面的边界。因此，该交集的边界由若干条边组成，它们分别是某张半平面边界线上的一段。这个交是凸集，故每条边界线只能为交集的边界至多贡献一条边。由此可知：n张半平面的公共交集是一个凸的多边形区域，其边界由至多n条边围成。图 4-8 给出了半平面相交的几种可能情况。该图中，各张半平面究竟位于其边界线的哪一侧，由深色阴影指示；而浅色阴影部分则为其公共交集。由 图 4-8 的(ii)和(iii)可见，其公共交集并不见得一定有界。另外，正如实例(iv)所说明的，其公共交集也可能退化为一条线段、一个点，或者象实例(v)那样根本就是空的。

下面就直截了当地给出一个分治式算法，计算任意 n 张半平面的公共交集。该算法利用了 IntersectConvexRegions 子程序，来计算两个凸多边形区域的交集。首先介绍该算法的全局结构。

算法 IntersectHalfplanes(H)

输入：由平面上 n 张半平面组成的一个集合 H

输出：凸多边形区域 $C := \bigcap_{h\in H} h$

1. **if** (card(H) == 1)
2. **then** C ← H 中唯一的那张半平面 h
3. **else** 将 H 分成两个子集 H_1 和 H_2，大小分别为$\lceil \frac{n}{2} \rceil$和$\lfloor \frac{n}{2} \rfloor$
4. C_1 ← IntersectHalfplanes(H_1)
5. C_2 ← IntersectHalfplanes(H_2)
6. C ← IntersectConvexRegions(C_1, C_2)

这样，似乎需要对IntersectConvexRegions子程序做进一步描述。不过且慢——在此之前的第 2 章中，难道不是已经遇到过这一问题吗？的确，〖推论 2.7〗曾经指出：可以在$O(n\log n + k\log n)$时间内，计算出任意两个多边形的交（其中，n为两个多边形所含顶点的总数）。然而，在将这一结果应用于当前这个问题的时候，还是需要格外小心——因为，此处所处理的区域可能是无界的，也可能会退化为一条线段或一个点。也就是说，这些区域并不见得是（严格意义上的）多边形。不过，只需稍做修改，第 2 章的那个算法就可以应用于当前的场合，而且这些改动并不困难。

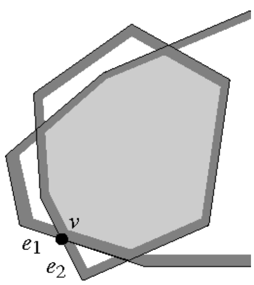


图4-9 C_1和C_2边界的交点数目

以下对这一方法做一分析。假设通过递归调用，我们已经计算出了两个区域C_1和C_2。既然二者分别都是由不超过$(\frac{n}{2}+1)$张半平面确定的，故它们各自都由至多$(\frac{n}{2}+1)$条边围成。利用第 2 章所介绍的算法，可以在$O((n+k)\log n)$时间内计算出它们之间的叠合部分（其中k为C_1各边与C_2各边之间的交点数目）。那么，k又是多少呢？考察分别来自C_1和C_2的两条边e_1和e_2，假定它们相交于点v（图 4-9）。无论e_1和e_2是以何种方式相交，其交点v必然会成为$C_1 \cap C_2$的一个顶点。反过来，既然$C_1 \cap C_2$是n张半平面的公共交集，其边界上至多含有n条边、n个顶点。由此可以看出，必然有$k \leq n$。于是，为了计算出C_1和C_2之间的交，需要花费的时间量为$O(n\log n)$。

这样，就得出了如下关于运行时间的递推关系：

$$T(n) = \begin{cases} O(1) & \text{如果 } n = 1 \\ O(n\log n) + 2T(\frac{n}{2}) & \text{如果 } n > 1 \end{cases}$$

其解为 $T(n) = O(n\log^2 n)$。

为得出上述结果，我们所使用的实际上是一个更加通用的子程序——它可以计算出任意两个多边形的交集。然而 INTERSECTHALFPLANES 算法所处理的每一个多边形区域都是凸的。既然如此，能否利用这一特点，得出一个更加高效的算法呢？正如马上就可看到的，这一问题的答案是肯定的。我们将假定，参与求交计算的区域都是二维的；其它的一些退化情况（比如其中之一是射线、线段或点，甚至二者都是如此），实际上更容易处理，这将做为一道习题留给读者。

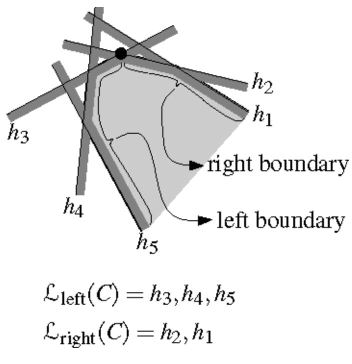


图4-10 C的边界可以由两组半平面共同描述

应该如何表示一个凸多边形（convex polygon）区域C呢？首先需要对此做出更为准确的定义。我们将C的边界划分为左、右两部分，将为它们贡献边的半平面相应地划分为两组，分别存储为一个有序表。如 图 4-10 所示，各半平面在各自列表中的存放次序，与自上而下遍历（左、右）边界时，各半平面边界线出现的次序一致。对应于左、右边界的列表分别记作$\mathcal{L}_{\text{left}}(C)$和$\mathcal{L}_{\text{right}}(C)$。边界上的顶点并不需要显式地记录下来——倘若需要，只要对相邻的边界线求交，即可得出对应的顶点。

为了简化对算法的描述，假定其中没有水平边。（为了能够处理水平边，可以这样约定：若某条水平边是从上方围住 C，则将它划归左边界；若它是从下方围住 C，则将其划归右边界。只要采取这种约定，就只需对下面将要介绍的算法做少量的调整。）

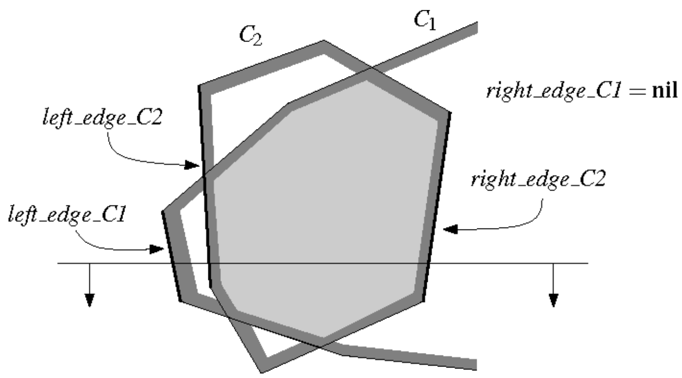


图4-11 扫描线算法维护（四条）边

与第 2 章中的那个算法一样，这里采用的也是一个平面扫描算法。我们将使用一条扫描线，自上而下地扫过整个平面，在此过程中，要动态维护C_1 和C_2 上与当前扫描线相交的那些边。既然C_1 和C_2 都是凸的，这样的边（在任何时刻都）不会超过 4 条。因此，不必采用复杂的数据结构来存储这些边；相反地，只需保留 4 个指针left_edge_C1、right_edge_C1、left_edge_C2 和right_edge_C2，分别指向它们。如果扫描线与某个区域的左边界或右边界不相交，那么对应的指针就被赋为nil。图 4-11 对各指针的定义做了说明。

应该如何对这些指针做初始化呢？令 C_1 的最高顶点的 y-坐标为 y_1；如果 C_1 有一条延伸到无穷远的无界边，就令 $y_1 = \infty$。类似地，也可以对 C_2 定义 y_2，并且令 $y_{start} = \min(y_1, y_2)$。为了计算出 C_1 与 C_2 的交集，我们可以将注意力限制在平面上 y-坐标不高于 y_{start} 的部分。这样，就将从 y_{start} 的高度启动扫描线——此时，指针 left_edge_C1、right_edge_C1、left_edge_C2 和 right_edge_C2 分别被赋为与直线 $y = y_{start}$ 相交的某条边。

在平面扫描算法中，通常还需要使用一个队列结构来存放事件。对当前这个问题来说，所谓的事件就是 C_1 和 C_2 的各边开始或者不再与扫描线相交的位置。这就意味着，应该检查与当前扫描线相交的所有边，从它们的下端点中挑出位置最高的那个，做为下一个事件点（event point）——而这一事件点又确定了将要处理的下一条边。（若这个 y-坐标高度上有多个端点，将按照从左到右的次序进行处理。若有两条边在这个端点处重合，则左边的那条优先处理。）因此，并不需要维护一个事件队列（event queue）——根据指针 left_edge_C1、right_edge_C1、left_edge_C2 和 right_edge_C2，完全可以在常数时间内确定下一个事件。

在每个事件点处，会有某条新的边 e 在边界上出现。为了处理边 e，首先要确定，e 到底是来自 C_1 还是 C_2；其次，还要确定它究竟是属于左边界，还是右边界。然后，才可以调用相应的子函数。

在此仅介绍其中的一种情况——e 属于 C_1 的左边界。其它的子程序都是类似的。

设 e 的上端点为 p。处理 e 的子程序，将找出 C 上可能存在的三种边：以 p 为上端点的边、以 e∩left_edge_C2 为上端点的边和以 e∩right_edge_C2 为上端点的边。具体的处理步骤是：

- 首先，检查 p 是否位于 left_edge_C2 和 right_edge_C2 之间。若是，则 e 必然为 C 贡献一条边，且该边起点为 p。于是，将以 e 所在直线为边界的那张半平面，加入到 L_{left}(C)中。
- 接下来，检查e是否与right_edge_C2 相交。若是，则其交点必然是C的一个顶点，而这两条边都各自为C贡献一条边。此时，有两种可能：若p位于right_edge_C2 的右侧，则p必然是它们所贡献的这两条边的起点（如 图 4-12(a)所示）；反之，若p位于right_edge_C2 的左侧，则p必然是其终点（如 图 4-12(b)所示）。

  若这两条边所贡献的边均起始于其交点，则必须将 e 所对应的那张半平面加入 L_{left}(C)中，而将 right_edge_C2 所对应的半平面加入 L_{right}(C)中。反之，若它们贡献的这两条边终止于其交点处，就什么都不需要做——因为，它们在早前必然已经以某种方式被发现过了。

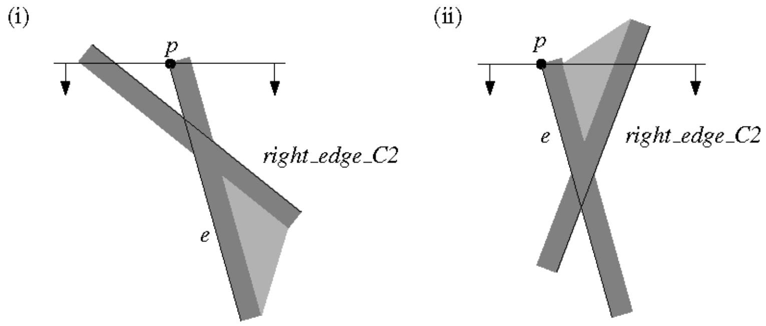


图4-12 e与right_edge_C2相交时的两种可能情况

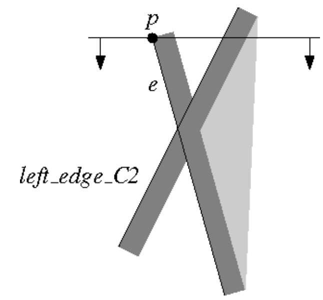


图4-13 e与left_edge_C2相交的情况

- 最后，要检查e是否与left_edge_C2 相交。若相交（如 图 4-13 所示），则其交点必然是C的一个顶点。在C上起始于该顶点的那条边，要么是e上的一段，要么是left_edge_C2 上的一段。究竟是哪种情况，可在常数时间内判断出来——若p位于left_edge_C2 左侧，则该边必是来自e上的一段；反之，必是来自left_edge_C2 上的一段。在确定了为C贡献这条边的

究竟是e还是left_edge_C2 之后，就可以将对应的那张半平面加入L_{left}(C)中。

需要注意的是，我们可能会将两张半平面加入到 L_{left}(C)中——它们分别以 e 和 left_edge_C2（所在的直线）为边界线。这种情况下，应该按照什么次序引入这两张半平面呢？只有当 left_edge_C2 在 C 上确定了起始于 left_edge_C2 与 e 交点的一条边时，才会加入 left_edge_C2。若与此同时也需要加入 e 所对应的那张半平面，则不外乎两种可能——e 在 C 上确定的那条边，要么起始于自己的上端点，要么起始于它与 right_edge_C2 的交点。无论是哪种情况，我们都必须首先加入 e 所对应的半平面——唯此才能确保上面给出的检查次序。

总而言之，可以在常数时间内处理每一条边，因此，也就可以 $O(n)$时间内计算出任意两个凸多边形的交集。为了说明算法的正确性，我们需要证明：该算法可以按照正确的次序，引入对应于 C 上各边的半平面。试考察 C 上的任一条边，令其上端点为 p。于是，p 要么是来自 C_1 或者 C_2 上某条边的上端点，要么是分别来自 C_1 和 C_2 的两条边 e 和 e'的交点。若是前一种情况，则在（扫描线）遇到 p 时，该算法必然会找出 C 上的这条边；若是后一种情况，则在（扫描线）遇到 e 和 e'的交点时，该算法也将找出这条边。因此，确定 C 上各边的所有半平面，都将被引入到边表中。而且，它们的加入次序必然是正确的——这一点不难证明。

可以将此归纳为下述结论：

〖定理 4.3〗

平面上任意两个凸多边形区域的交集，都可以在 $O(n)$时间内计算出来。

这个定理意味着，IntersectHalfplanes 算法的合并阶段只需花费线性的时间。于是，该算法运行时间的递推关系就是：

$$T(n) = \begin{cases} O(1) & \text{如果} n = 1 \\ O(n) + 2T(\frac{n}{2}) & \text{如果} n > 1 \end{cases}$$

由此可以得出如下结论：

〖推论 4.4〗

给定平面上的一组共 n 张半平面，可以使用线性的空间，在 $O(n\log n)$时间内计算出其公共的交集。

计算一组半平面公共交集的问题，与计算凸包的问题紧密相关，因此还可以给出另一个算法。与第 1 章所介绍的ConvexHull算法相比，这个算法几乎是一样的。第 8.2 节和第 11.4 节还将对半

平面的交集与凸包之间的关系做详细的讨论。相对本书后面的各章，这两节是独立的，因此如果读者对这方面感兴趣，完全可以直接阅读。

4.3 递增式线性规划

前一节介绍了一种方法，计算任意 n 张半平面的公共交集。也就是说，给定一组共 n 个线性约束条件，可以计算出同时满足它们的所有解。该算法的运行时间为 $O(n\log n)$。可以证明，这已经是最优的了——与排序算法一样，任何可以对一组半平面进行求交的算法，在最坏情况下必须花费 $\Omega(n\log n)$时间。然而，铸造问题的要求与此不同——给定一组线性约束条件，我们并不需要得到整个解集；实际上，只要得到一个可行解即可。这样，就有可能设计出更快的算法。

找出满足一组线性约束条件的一个解，与运筹学（operations research）中的一个著名问题密切相关，这就是所谓的线性优化（linear optimization）问题，或者称作线性规划（Linear Programming）问题。（这里的“linear programming”一词很早就造出来了，而后来的“programming”一词却有特定的含义——为计算机编制指令。）这两个问题的区别在于：线性规划的目标，是要在一组约束条件的可行解域中，找出一个特定的解——具体地讲，这个解将使一个定义在有关变量上的函数极大化。更准确地，每个线性优化问题都可以表述为如下形式：

在满足约束条件：

$$
\begin{aligned}
a_{1,1}x_1 + \dots + a_{1,d}x_d &\leq b_1 \\
a_{2,1}x_1 + \dots + a_{2,d}x_d &\leq b_2 \\
&\vdots \\
a_{n,1}x_1 + \dots + a_{n,d}x_d &\leq b_n
\end{aligned}
$$

的前提下，使函数： $c_1x_1 + c_2x_2 + \dots + c_dx_d$ ***极大化***

在这里，实数c_i、$a_{i,j}$和b_i共同构成了问题的输入。待极大化的那个函数称作目标函数（objective function），而约束条件集与目标函数合在一起，就是一个线性规划问题（linear program）。其中所含变量的个数d，称作线性规划问题的维数（dimension of the linear program）。我们已知，每个线性约束条件都可看作$\mathbb{R}^d$中一个半空间。所有同时满足这些约束条件的点，构成了这些半空间的交集，称作线性规划问题的可行解域（feasible region）。落在其中的点（解）称作是可行的（feasible），而落在其外的点则称作是不可行的（infeasible）。回顾 图 4-8 我们还记得，可行解域有可能无界，甚至可能为空。若是后一情形，对应的线性规划问题也称作是不可行的（infeasible）。至于目标函

数，则可以视作$\mathbb{R}^d$空间中的某个方向——所谓“使函数$c_1x_1 + c_2x_2 + \dots + c_dx_d$极大化”，即沿方向$\vec{c} = (c_1, \dots, c_d)$ 找到一个极点。这样，如 图 4-14 所示，所谓一个线性规划问题的解，就是在可行解域中沿着方向$\vec{c}$的那个极值点（若存在的话）。对应于方向$\vec{c}$的那个目标函数，记作f_c。

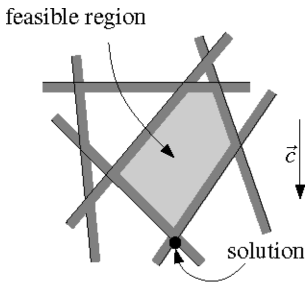


图4-14 线性规划问题的解就是可行解域中沿特定方向极值点

许多运筹学问题都可描述为线性规划的形式，因此运筹学也就此做了大量研究，并得出了许多线性规划算法，它们在实际应用中也的确行之有效——如著名的单纯形法（simplex algorithm）。

还是回到我们的问题。我们面对的是一组双变量的约束条件，而目标则是找出满足这组约束条件的一个解。为此，可以任取一个目标函数，然后以这个目标函数以及给定的线性约束条件做为输入，求解一个线性规划问题。为完成这一任务，可采用单纯形法，或运筹学所给出的任一线性规划算法。然而，此处的问题与通常的一般性问题很不一样——在运筹学中，无论是约束条件还是其中的变量，数目都很大；而当前的问题中，变量只有两个。在求解这类低维线性规划（low-dimensional linear programming）问题时，传统的线性规划方法反而效率不高；而在计算几何（computational geometry）中发展起来的一些方法，却更加行之有效——下面将要介绍的算法，就是一例。

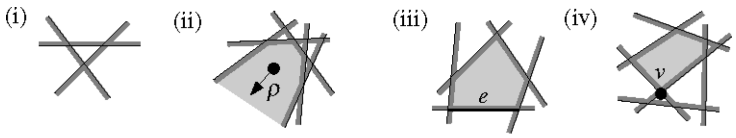


图4-15 线性规划的解有四种可能的情况

针对这里的二维线性规划问题，我们将其中的n个线性约束条件合起来，记作集合H。对应于目标函数的矢量记作$\vec{c} = (c_x, c_y)$；这样，目标函数就是$f_c(p) = c_xp_x + c_yp_y$。我们的目标就是找出一个点$p \in \mathbb{R}^2$，满足$p \in \cap H$并且使$f_c(p)$达到最大。我们将该线性规划记作$(H, \vec{c})$，并且用C来表示其对应的可行解域。一个线性规划$(H, \vec{c})$的解，有四种可能的情况。图 4-15 画出了这四种情况。在这些例子中，对应于目标函数的矢量，都是垂直朝下的。

(i) 待解的线性规划问题是不可行的，亦即，对应的约束条件集是空的。

(ii) 沿着方向$\vec{c}$，可行解域是无界的。此时，存在某条射线ρ完全落在可行解域 C 中——沿该射线，函数 f_c的取值可以任意增长。果真如此，我们就要求算法描述出这样的一条射线。

(iii) 可行解域的边界上有一条边 e，其外法矢的方向与$\vec{c}$相同。在这种情况下，虽然该线性规划问题是有解的，却不唯一——实际上，在 e 上的任一点处，函数 $f_c(p)$都达到了最大。

(iv) 只要不是前面的那三种情况，就肯定有解，而且解是唯一的——这个解是 C 的某个顶点，而且这个顶点是沿着方向$\vec{c}$的极点。

我们的二维线性规划算法，是一个递增式算法。它逐一引入各个约束条件，始终维护当前的最优解。不过，对于每一当前最优解，该算法不仅要求它是定义明确的，而且还应是唯一的。也就是说，该算法假定：每一步迭代所对应的可行解域，都属于上述第 (iv)种情况——存在唯一的最优顶点。

为满足这一要求，可以在待解的线性规划问题中引入两个附加的约束条件，以确保其有界性。比如，若 $c_x > 0$ 且 $c_y > 0$，就增加约束条件 $p_x \le M$ 和 $p_y \le M$，其中 $M \in \mathbb{R}$ 是一个很大的数。我们的想法是，若原来的线性规划问题有界，则只要 M 选得足够大，就不致于对最优解有任何影响。

在线性规划的很多实际应用中，这类附加条件本来就是某种自然的限制。以铸造问题为例，受机械上的限制，沿任何接近水平的方向，都不可能移出多面体。比如说，无法沿着与 xy-平面夹角小于 1 度的方向，将多面体取出。根据这类限制条件，可以立即得出 p_x 和 p_y 绝对值的上限。

那么，如何才能正确地判别出无界线性规划问题（unbounded linear program）呢？此外，能否不对解设置人为的强制约束条件，即对有界的问题进行求解呢？第 4.5 节将讨论这些问题。

为了精确起见，需要明确地对这两个新引入的约束条件做出定义：

$$m_1 := \begin{cases} p_x \le M & 若 c_x > 0 \\ -p_x \le M & 否则 \end{cases}$$

$$m_2 := \begin{cases} p_y \le M & 若 c_y > 0 \\ -p_y \le M & 否则 \end{cases}$$

请注意，m_1 和 m_2 只是做为$\vec{c}$的一个函数，它们都与 H 中具体的半平面无关。可行解域 $C_0 = m_1 \cap m_2$ 是一个正交的楔形区域。

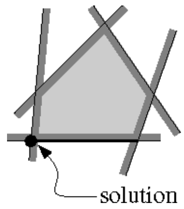


图4-16 按照字典序消除退化情况的歧义性

情况(iii)的解也可以认为是唯一的。如 图 4-16 所示，为此，需要做另一个简单的约定：如果同时有多个最优解，我们就按照字典序取其中的最小者。这一约定的效果，就相当于（假想着）对$\vec{c}$做了极小角度的旋转，从而使之不再与任何半平面（的边界）垂直。

不过，在做如此处理时需要格外小心——因为，即便是有界线性规划问题（bounded linear program），按照字典序也不见得必定存在一个最小的解（参见 习题 4.11）。只有在强加了m_1和m_2这两个约束条件，才能保证这种情况不致发生。

只要采用了上述两个约定，就可以保证每个线性规划问题都是可行的，解也是唯一的，而且这个解必然是可行解域的某个顶点。这个顶点称作最优解顶点（optimal vertex）。

任取一个线性规划问题(H, $\vec{c}$)。我们将其中的半平面依次编号为：$h_1, h_2, \ldots, h_n$。其中的前 i 个约束条件，加上专门附加的两个约束条件，就构成了集合 H_i；令 H_i 所对应的可行解域为 C_i：

$$H_i := \{m_1, m_2, h_1, h_2. \ldots, h_i\}$$

$$C_i := m_1 \cap m_2 \cap h_1 \cap h_2 \cap \ldots \cap h_i$$

这样，只要对某个 i 有 $C_i = \varnothing$，则对任何 $j \geq i$，都有 $C_j = \varnothing$——也就是说，该线性规划问题是不可行的。因此，某个线性规划问题一旦成为不可行的，我们就可以立即终止算法。

在引入下一张半平面 h_i 时，最优解顶点将会如何变化？下面的这则引理回答的正是这个问题。该引理建立在我们的算法之上。

〖引理 4.5〗

设 $1 \leq i \leq n$，而且 C_i 和 v_i 的定义如上。则有

(i) 若 $v_{i-1} \in h_i$，则 $v_i = v_{i-1}$；

(ii) 若 $v_{i-1} \notin h_i$，则要么 $C_i = \varnothing$，要么 $v_i \in l_i$（其中 l_i 就是 h_i 的边界线）。

〖证明〗

(i) 设$v_{i-1} \in h_i$。因为$C_i = C_{i-1} \cap h_i$，而且$v_{i-1} \in C_{i-1}$，所以必有$v_{i-1} \in C_i$。此外，既然$C_i \subseteq C_{i-1}$，故与原先C_{i-1}的最优解顶点相比，C_i的最优解顶点不可能更优[①]。因此，v_{i-1}必然还是C_i的最优解顶点。

(ii) 设 $v_{i-1} \notin h_i$。假设引理不成立，也就是说，C_i不为空，同时 v_i 也不落在 l_i 上。

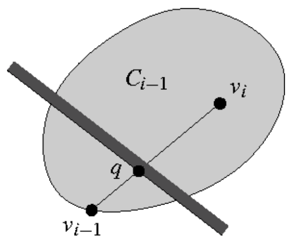


图4-17 $v_{i-1} \notin h_i$的情况

如图 4-17 所示，考察线段$\overline{v_{i-1}v_i}$。首先，我们已知$v_{i-1} \in C_{i-1}$；既然$C_i \subseteq C_{i-1}$，故必有$v_i \in C_{i-1}$。考虑到C_{i-1}的凸性，线段$\overline{v_{i-1}v_i}$必然整体包含于C_{i-1}之中。因为v_{i-1}是C_{i-1}的最优解顶点，而且目标函数 $f_{\vec{c}}$是线性的，所以在沿着$\overline{v_{i-1}v_i}$从v_i到v_{i-1}的运动过程中，$f_{\vec{c}}(p)$必然是单调递增的。再考虑$\overline{v_{i-1}v_i}$与l_i的交点q。既然$v_{i-1} \notin h_i$而$v_i \in C_i$，故这个交点必然存在。根据前面的分析，线段$\overline{v_{i-1}v_i}$包含于C_{i-1}之中，因此点q必然属于C_i。由于目标函数沿着$\overline{v_{i-1}v_i}$是单调递增的，所以$f_{\vec{c}}(q) > f_{\vec{c}}(v_i)$——这与$v_i$的定义不合。 ❑

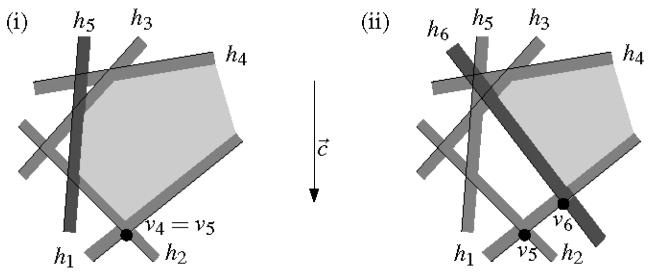


图4-18 引入下一张半平面

在引入新的半平面时，有两种可能的情况，图 4-18 画出了这两种情况。如 图 4-18(a)所示，在已经引入了前四张半平面之后，对应的最优解顶点为v_4；继续加入下一张半平面h_5，将发现v_4落在h_5之中。这种情况下，最优解顶点保持不变。然而接下来再加入h_6 后，发现它不再包含此前的最优解顶点。在这种情况下，就需要计算出一个新的最优解顶点。根据 〖引理 4.5〗，新的最优解顶点v_6必然落在h_6的边界线上（如 图 4-18(b)所示）。那么，具体地如何才能找出这个新的最优解顶点呢？〖引理 4.5〗并没有回答这个问题。所幸的是，正如马上就要说明的，这个问题并不困难。

① 随着各半平面的不断引入，目标函数必然单调非增。——译者

现假设当前的最优解顶点 v_{i-1} 没有落在下一张半平面中。此时，可以如此表述待求解的问题：

在满足约束条件“对任何 $h \in H_{i-1}$ 都有 $p \in h$”的前提下，在直线 l_i 上找出一点 p，使函数 $f_{\vec{c}}(p)$ 取最大值。

为了简化这些术语，假设 l_i 是非垂直的——这样，就可以通过 x-坐标来对其做参数化描述。于是，可以定义出一个函数：

$$\overline{f_{\vec{c}}} : R \mapsto R$$

对于任何点 $p \in l_i$，这个函数都满足 $f_{\vec{c}}(p) = \overline{f_{\vec{c}}}(p_x)$。对任意一张半平面 h，将 h 的边界线与 l_i 之间交点的 x-坐标记作 $\sigma(h, l_i)$。（若这个交点不存在，则要么 l_i 上的任何点都满足约束条件 h，要么 l_i 上的任何点都不满足约束条件 h。若是前一种情况，则这个约束条件可以忽略不计；若是后一种情况，则可以立即报告“这个线性规划问题是不可行的”。如 图 4-19 所示，关于解的 x-坐标，我们可以得出一个约束条件，其形式或者是 $x \geq \sigma(h, l_i)$，或者是 $x \leq \sigma(h, l_i)$。具体为哪种形式，取决于 $l_i \cap h$ 在左侧有界，还是在右侧有界。

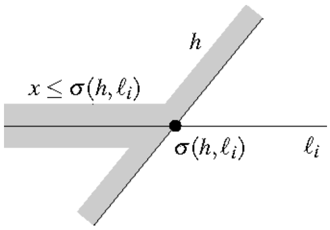


图4-19 x-坐标的约束条件

这样，我们就可以将问题重新表述为如下形式：

在满足

$x \geq \sigma(h, l_i)$ **（当 $h \in H_{i-1}$ 而且 $l_i \cap h$ 的左侧有界时），或者**

$x \leq \sigma(h, l_i)$ **（当 $h \in H_{i-1}$ 而且 $l_i \cap h$ 的右侧有界时）**

的前提下，使函数 $\overline{f_{\vec{c}}}$ 极大化

这是个一维线性规划问题，可以很快解答。令

$$x_{left} = \max_{h \in H_{i-1}} \{\sigma(h, l_i) : l_i \cap h \text{ 在左侧有界}\}$$

以及

$$x_{right} = \min_{h \in H_{i-1}} \{\sigma(h, l_i) : l_i \cap h \text{ 在右侧有界}\}$$

则区间$[x_{left} : x_{right}]$就是该一维线性规划问题的可行解域。因此，如果 $x_{left} > x_{right}$，对应的线性规划问题就是不可行的；否则，最优解顶点必然存在于 l_i 上，而且它或者对应于 x_{left}，或者对应于 x_{right}——具体是哪个点，取决于具体的目标函数。

需要指出的是，由于约束条件 m_1 和 m_2 的引入，这种一维线性规划问题将不可能是无界的。

这样，我们就得出了如下引理。

〖引理 4.6〗

每个一维线性规划问题都可在线性时间内求解。因此，若出现〖引理 4.5〗中的情况(ii)，则在$O(i)$时间内，要么可以计算出更新后的最优解顶点v_i，要么可以判断出这个线性规划问题是不可行的。

接下来更具体地介绍线性规划问题的算法。与此前一样，依然用 l_i 表示半平面 h_i 的边界线。

算法 2DBOUNDEDLP(H, $\vec{c}$, m_1, m_2)

输入：一个线性规划问题$(H \cup \{m_1, m_2\}, \vec{c})$，其中 H 为 n 张半平面所组成的一个集合
$\vec{c} \in \mathbb{R}^2$
m_1 和 m_2 为解设定了边界

输出：若$(H \cup \{m_1, m_2\}, \vec{c})$是不可行的，则报告这一情况
否则，报告出使 $f_{\vec{c}}(p)$达到最大的（依字典序的）最小点

```
1.  令 v0 为 C0 的角点
2.  令 h1, h2, ..., hn 为 H 中的各张半平面
3.  for i ← 1 to n
4.      do if (v(i-1) ∈ hi)
5.           then vi ← v(i-1)
6.           else vi ← 点 p
                  (* p 来自 li 上，它满足 H(i-1) 中的所有约束条件，而且使 fc(p)达到最大 *)
7.                if (点 p 不存在)
8.                  then 报告“该线性规划问题不可行”，然后退出
9.  return vn
```

以下分析该算法的性能。

〖引理 4.7〗

算法 2DBoundedLinearLP 可以使用线性的空间，在 $O(n^2)$时间内，求解任何一个含有 n 个约束条件的双变量有界线性规划问题。

〖证明〗

为证明该算法能够正确地得出解答，需要证明，在经过每次迭代——也就是每引入一张新的半平面h_i——之后，点v_i总是C_i的最优解顶点。根据〖引理 4.5〗，可以立即证明这一点。倘若在l_i上的一维线性规划问题是不可行的，则C_i必是空集，于是$C = C_n \subseteq C_i$也必是空集——这说明，该线性规划问题是不可行的。

至于该算法只需要线性的空间这一点，可以很容易证明。我们逐一引入各张半平面，总共迭代 n 轮。其中第 i 轮所消耗的时间，主要花在算法的第 6 行上。因为这项工作是求解一个一维线性规划问题，所以需要的时间量为 $O(i)$。这样，总共需要的时间量不会超过

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

这正是本引理的结论。 ❑

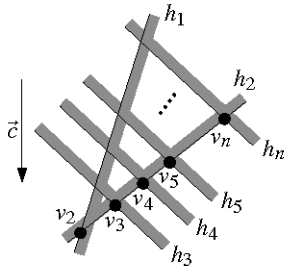


图4-20 最坏情况

尽管我们的线性规划算法简单得很，但是其运行时间却令人失望——即使是与此前所介绍的那个计算整个可行解域的算法相比，它的速度也要慢很多。那么，我们对时间性能的分析是否过于粗糙呢？每一轮迭代所需的时间，我们都是用$O(i)$来估计。毕竟，这个上界（upper bound）并不总是紧的——只有在$v_{i-1} \in h_i$的时候，第i轮迭代才需要$\theta(i)$时间；而要是$v_{i-1} \in h_i$，则只需常数时间。因此，只要能够界定最优解顶点发生变化的次数，或许能够得出一个更好的运行时间复杂度。然而不幸的是，（在最坏情况下）最优解顶点的确可能需要改变n次——对于某些半平面集，的确存在某些次序，按照这种次序每引入一张新的半平面，都会使此前的最优解顶点不再是最优的。图 4-20 就是一个例子。此时该算法确实需要运行$\Theta(n^2)$时间。那么，如何才能避开这类讨厌的情况呢？

4.4 随机线性规划

以上所举的例子中，最优解顶点需要改变 n 次。重新审视这一例子后我们可能会意识到，问题的关键并不在于做为输入的半平面集。若按照另一次序（比如 $h_n, h_{n-1}, \ldots, h_3$）引入各张半平面，则尽管在加入 h_n 时最优解顶点的确需要改变，但此后再也不会改变。读者可能会猜测：对任一半平面集 H，都存在某种“好”的处理次序。这个猜想正确与否？答案是肯定的。然而，这依然于事无补。即使这样的次序的确存在，要想具体地找出这样一个次序也绝非易事。毕竟，这样的一个次序必须在算法的一开始就找出——然而此时，我们对各半平面之间相交的情况还一无所知。

这样，就出现了一个绕有趣味的现象。我们的确找不到任何方法，可以（在事先）确定 H 的一个次序，以保证运行的时间性能足够好。尽管如此，只要通过一种非常简单的方法，就可以让我们走出困境。这个方法就是，直接采用 H 的一种随机次序（random ordering）。自然地，按照这种策略，我们的运气还是有可能会很糟，以至于采用的次序依然会导致平方量级的运行时间。然而要是运气好的话，我们选用的次序也会使算法运行得更快。事实上，下面马上就将证明，大多数的次序都会导致一个快速的计算过程。为了保持完整性，首先将该算法“复述”一遍：

算法 2DRANDOMIZEDBOUNDEDLP(H, $\vec{c}$, m_1, m_2)

输入：一个线性规划问题(H∪{m_1, m_2}, $\vec{c}$)
(* 其中 H 为由 n 张半平面组成的一个集合，$\vec{c} \in \mathbb{R}^2$，m_1 和 m_2 为解设定了边界 *)

输出：若(H∪{m_1, m_2}, $\vec{c}$)是不可行的，则报告这一情况
否则，报告出使 $f_{\vec{c}}(p)$达到最大的（依字典序的）最小点

1. 令 v_0 为 C_0 的角点
2. 调用 RANDOMPERMUTATION(H[1...n])，生成这些半平面的一个随机排列：$h_1, h_2, \ldots, h_n$
3. **for** i ← 1 **to** n
4. **do if** ($v_{i-1} \in h_i$)
5. **then** $v_i \leftarrow v_{i-1}$
6. **else** $v_i \leftarrow$ 点 p
   (* p 来自 ℓ_i 上，它满足 H_{i-1} 中的所有约束条件，而且使 $f_{\vec{c}}(p)$达到最大 *)
7. **if** (点 p 不存在)
8. **then** 报告“该线性规划问题不可行”，然后退出
9. **return** v_n

与前面的那个算法相比，只有第 2 行不同——这里，在开始逐一引入各张半平面之前，首先随机地打乱它们的次序。为此需要假设：我们可以使用某个现成的随机数发生器（random number generator）。对于任一整数k，这个发生器都能够在常数时间内，给出介于 1 到k之间的一个随机整

数Random(k)。这样，采用如下算法，就可以在线性时间内生成一个随机排列次序[①]。

```
算法 RandomPermutation(A)
输入：数组 A[1…n]
输出：由输入数组中各元素组成的另一个数组 A[1…n]，其中各元素的次序已经随机打乱
1.   for k ← n downto 2
2.      do rndindex ← Random(k)
3.         Exchange(A[k], A[rndindex])
```

这个新的算法，称为随机算法（randomized algorithm）。也就是说，其运行时间具体是多少，取决于它所做出的一系列随机选择。（就目前的线性规划算法而言，其中的随机选择是在子程序RandomPermutation 中进行的。）

在将我们的递增式线性规划算法改写成随机式版本之后，算法的执行时间将是多少呢？这一问题不易回答。运行时间具体多少，完全取决于第 2 行所计算出来的次序。考察由 h 张半平面组成的一个固定的集合 H。第 2 行确定了哪种次序，2DRandomizedBoundedLP 就会按照该次序来处理其中各张半平面。既然 n 个对象可能的排列共有 n!种，算法也就相应地有 n!种执行的方式，而每一种方式也各有其不同的运行时间。排列是随机选取的，故各种运行时间出现的可能性也相同。因此，需要分析该算法的期望运行时间（expected running time）——也就是全部共 n!种可能排列次序的平均运行时间（average running time）。以下引理指出，我们的随机线性规划算法的期望运行时间为$O(n)$。这里对算法的输入并没有做任何假设，认识到这一点很重要。这里所说的“期望”，是相对于对各半平面进行处理的随机次序而言；而且，这个结论对任何半平面集都成立。

〖引理 4.8〗

任一包含 n 个约束条件的二维线性规划问题，都可以在 $O(n)$的期望运行时间内得到解答；而且，所需要的空间在最坏情况下也不会超过线性规模。

〖证明〗

正如此前已经看到的，该算法所需要的空间必是线性的。

调用子函数 RandomPermutation 的时间总共为 $O(n)$，因此下面只需要对引入半平面 $h_1, \ldots, h_n$ 所花费的时间进行估计。每加入一张半平面，若最优解顶点无需改变，则只需常数的时间。

[①] 这一结论在理论上的确成立，但在实践中却值得商榷。通常的随机数生成算法，本质都是一样的：取 rand(x) = (a + b × x) mod p，其中 a 为随机种子，b 为固定的步长，p 为足够大的素数。因此，按照这种方式生成的“随机”排列序列，将完全取决于随机种子 a 的选取。然而实际上，只要元素的数目 n > 20，可能的排列次序就将多达 $n! > 2^{64}$ 种。也就是说，即使采用 64 位无符号的整数，也无法覆盖所有可能的序列。——译者

要是最优解顶点有所变化，我们就需要去求解一个一维线性规划问题。那么，求解所有这类一维线性规划问题所需的时间，总共是多少呢？现在，我们就来对此做一界定。

取随机变量 X_i，若 $v_{i-1} \notin h_i$，其取值就为 1，否则为 0。你应该记得，包含 i 个约束条件的一维线性规划问题的求解需要 $O(i)$时间。因此，所有半平面总共消耗在第 6 行上的时间就是

$$\sum_{i=1}^{n} O(i) \cdot X_i$$

为了界定上述和式的期望值，需要利用期望的线性律（linearity of expectation）——一组随机变量总和的期望值，等于这些随机变量各自期望值的总和。即便其中的随机变量不是相互独立的，该性质也依然满足。这样，为求解这些一维线性规划问题，总体运行时间的期望值为

$$E\left[\sum_{i=1}^{n} O(i) \cdot X_i\right] = \sum_{i=1}^{n} O(i) \cdot E[X_i]$$

那么，$E[X_i]$又是多少呢？它正好等于 $v_{i-1} \notin h_i$ 的概率。以下就来分析这个概率。

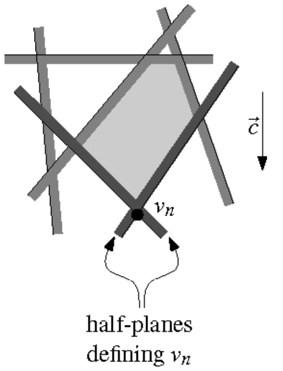


图4-21 由C_n到C_{n-1}的后向分析

这里将采用一种称作“后向分析”（backward analysis）的技术——也就是说，“倒过来”考察这个算法。如 图 4-21 所示，假设算法已经运行结束，并计算出了最优解顶点v_n。既然v_n是C_n的一个顶点，它肯定是由至少两张半平面（的边界线）确定的。现在，将时间倒退一步，反过来考察C_{n-1}。我们注意到，可以认为C_{n-1}是从C_n得到的——为此，只需将h_n删去。在这一过程中，最优解顶点会发生什么变化呢？这个点如果发生变化，只有一种可能——v_n不是C_{n-1}沿着$\vec{c}$方向的那个极大顶点。也就是说，h_n必然是参与确定v_n的半平面之一。然而，所有半平面都是按照随机次序引入的，因此，h_n只是$\{h_1, h_2, \ldots, h_n\}$中随机的一员。这样，$h_n$是确定$v_n$的半平面之一的概率，至多为$\frac{2}{n}$。

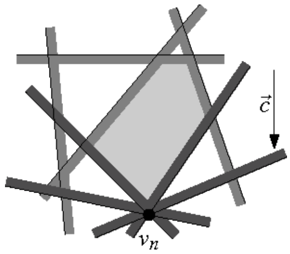


图4-22 多张半平面的边界同时穿过v_n

为什么说是“至多“呢？首先，如 图 4-22 所示，可能有多于两张半平面的边界同时穿过v_n。其中，有两张半平面包含了与v_n相关联的两条边。在这种情况下，无论删除这两张半平面中的那一张，都不会对v_n有任何影响。此外，v_n也有可能是由m_1和m_2定义的，而它们都不是h_n的n个随机候选。在这两种情况中，上述概率都严格地小于$\frac{2}{n}$。

以上分析过程，适用于算法的每一步——为了界定 $E[X_i]$，我们可以固定前 i 张半平面。这样，就确定了 C_i。而为了对此前最后一步（亦即加入 h_n 时）的情况进行分析，我们可以再次地反过来思考。在引入 h_i 后，需要重新计算一个新的最优解顶点的概率是多少呢？这个概率，正好等于在将某张半平面从 C_i 中删除之后，最优解顶点发生改变的概率。后一事件的发生，只有一种可能——被删除的半平面，是来自于已经被固定了的集合$\{h_1, ..., h_n\}$的至多两张特定半平面之一。这些半平面都是按照随机次序引入的，故 h_i 是这种特定半平面之一的可能性至多为$\frac{2}{i}$。当然，得出这一概率值的前提是，前 i 张半平面构成 H 的某个固定的子集。不过反过来，只要是一个固定的子集，就必然可以得出这一上界——亦即，该上界是无条件成立的。于是，就有$E[X_i] \leq \frac{2}{i}$。现在，对于一维线性规划问题求解的期望运行时间，我们可以给出如下上界：

$$\sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n)$$

至于算法其余部分所消耗的时间，根据我们在此前的分析，也是 $O(n)$。 ❑

需要再次强调的是，这里所指的“期望”，只是相对于算法可能做出的各种随机选择而言。我们并不是对所有可能的各种输入情况进行平均。总而言之，对于由 n 张半平面组成的任何一个输入集，该算法的期望运行时间都是 $O(n)$——也就是说，没有哪个输入集是“坏”的。

4.5 无界线性规划问题

在前几节中，为了回避无界线性规划问题，我们人为地引入了两个附加约束条件。这种方法并

不总是能够奏效。就算待解的线性规划问题是有界的，我们也可能不知道多大的 M 才足以使之有界。此外，在实际应用中，无界线性规划问题的确会出现，对于这类问题，我们也必须能够正确地解答。

首先来看看，如何才能判别某一线性规划问题$(H, \vec{c})$是否无界。正如此前已知，这意味着存在某条射线ρ，它完全包含在该问题的可行解域当中——于是沿着该射线，函数 f_c的取值可以任意增长。

若将该射线的起点记作 p，其方向记作矢量$\vec{d}$，则可以得出ρ的参数化描述如下：

$$\rho \quad = \quad \{ p + \lambda\vec{d} : \lambda > 0 \}$$

函数 f_c的取值可任意增长，当且仅当$\vec{d}\cdot\vec{c} > 0$。另一方面，任一 $h \in H$ 都对应于两条法矢，它们以边界线为基准，分别指向两侧；若将其中指向可行解域一侧的那条法矢记作$\vec{\eta}$，则还有$\vec{d}\cdot\vec{\eta}(h) \geq 0$。以下引理将指出：在判断一个线性规划问题是否无界时，这两个条件不仅必要，而且也充分。

〖引理 4.9〗

一个线性规划问题是无界的，当且仅当存在某个矢量$\vec{d}$满足$\vec{d}\cdot\vec{c} > 0$ 的，同时使得对于任何 $h \in H$，不仅有$\vec{d}\cdot\vec{\eta}(h) \geq 0$，而且若令 $H' = \{h \in H : \vec{\eta}(h)\cdot\vec{d} = 0\}$，则线性规划问题$(H', \vec{c})$总是可行的。

〖证明〗

根据上面的分析，“仅当”的方向可以直接得证。因此，只需证明“当”的方向。

我们来考察任意一个线性规划问题$(H, \vec{c})$以及满足本引理条件的一个矢量$\vec{d}$。既然$(H', \vec{c})$是可行的，则其中必然存在一个点 $p_0 \in \cap_{h \in H'} h$。现在取射线$\rho_0 := \{p_0 + \lambda\vec{d} : \lambda > 0\}$。因为对任何 $h \in H'$都有$\vec{d}\cdot\vec{\eta}(h) = 0$，所以射线$\rho_0$必然完全被包含在每张 $h \in H'$之内。此外，由于$\vec{d}\cdot\vec{c} > 0$，故沿着射线ρ_0，函数 f_c的取值必然可以任意地增长。

对任一 $h \in H \backslash H'$，都有$\vec{d}\cdot\vec{\eta}(h) > 0$。这就说明，必然存在某个参数$\lambda_h$，使得对任何的$\lambda \geq \lambda_h$，都满足 $p_0 + \lambda\vec{d} \in h$。现在，令$\lambda' := \max_{h \in H \backslash H'} \lambda_h$，令 $p := p_0 + \lambda'\vec{d}$。于是，我们就知道射线

$$\rho = \{ p + \lambda\vec{d} : \lambda > 0\}$$

必然完全落在每张半平面 $h \in H$ 之内，这就意味着，$(H, \vec{c})$是无界的。 ❑

这样，我们就可以仿照第 4.1 节的处理方法，判断出任意给定的二维线性规划问题$(H, \vec{c})$是否无界，并进而求解一个一维线性规划问题。

首先，对坐标系进行旋转，以使$\vec{c}$垂直向下——即$\vec{c} = (0, 1)$。满足$\vec{d}\cdot\vec{c} > 0$的任一方向矢量$\vec{d} = (d_x,$

d_y)，都可以规范化为$\vec{d} = (d_x, 1)$的形式，并进而表示为直线 $y = 1$ 上的一个点 d_x。给定一个单位矢量 $\vec{\eta}(h) = (\eta_x, \eta_y)$，不等式

$$\vec{d}\cdot\vec{\eta}(h) = d_x\eta_x + \eta_y \geq 0$$

可以转化为另一个不等式：$d_x\eta_x \geq -\eta_y$。于是就得到了一组共 n 个线性不等式——或换而言之，一个一维线性规划问题$\overline{H}$。（实际上，这的确有滥用术语之嫌——因为做为一个线性规划问题，除了一组约束条件之外，还应该有一个目标函数。不过，鉴于我们在这里并不在乎其可行性，暂时忽略目标函数反而会更为方便。）

若$\overline{H}$有一个可行解 d_x^*，则定义一个子集 $H'\subseteq H$，对于其中的每张半平面 h，这个解都是紧的——亦即，必须满足 $d_x^*\eta_x + \eta_y = 0$。我们仍然需要确认，不等式组 H'是可行的。如此一来，岂不是又回到了二维线性规划问题吗？的确如此，不过，这是一个很特别的问题——对于每一张 $h \in H'$，其法矢$\vec{\eta}(h)$都与 $d = (d_x^*, 1)$垂直。这就意味着，h 的边界线必然与$\vec{d}$平行。换而言之，H'中所有半平面的边界线都相互平行，因此在用 x-坐标轴对它们求交之后，将再次得到一个一维线性规划问题$\overline{H'}$。若$\overline{H'}$是可行的，则最初的线性规划问题必是无界的，而且正如在上述引理（的证明）中那样，可以在 $O(n)$ 时间内，构造出一条可行的射线ρ。反之，若$\overline{H'}$是不可行的，则 H'也不可行，因而 H 也不可行。

如果$\overline{H}$没有一个可行解，那么根据上述引理，最初的那个线性规划问题$(H, \vec{c})$必然是有界的。此外，在这种情况下，我们是否还可以得出更多的信息呢？你应该还记得一维线性规划问题的解（的判断准则）——H 是不可行的，当且仅当那些左侧有界的射线中边界的最大值，要大于那些右侧有界的射线中边界的最小值。如果将这两条射线分别记作$\overline{h_1}$和$\overline{h_2}$，这个充要条件就等价于"$\overline{h_1}$和$\overline{h_2}$的交集为空"。若将对应于这两个约束条件的那两张原始的半平面分别设为 h_1 和 h_2，则该充要条件可以进一步等价于"$(\{h_1, h_2\}, \vec{c})$是有界的"。因此，我们可以将 h_1 和 h_2 称作"凭证"——因为，它们"证明"了$(H, \vec{c})$的确是有界的。

这种凭证有多大作用？若能够注意到下面这个事实，你就应该对其作用看得很清楚了：只要找到这样的两个凭证 h_1 和 h_2，就可以象在 2DRANDOMIZEDBOUNDEDLP 中使用 m_1 和 m_2 那样对它们加以利用。这意味着，我们不再需要人为地去设置条件，以对解的允许范围进行强制性限定。

同样地，在此也必须格外细心。有一种特殊情况是：虽然线性规划问题$(\{h_1, h_2\}, \vec{c})$是有界的，但是按照字典序，却不存在一个最小的解。当这里的一维线性规划问题因为单一的某个约束条件 h_1 而不可行时（具体地，也就是当$\vec{\eta}(h_1) = -\vec{c} = (0, -1)$时），就会发生这种情况。如果遇到这种情况，可以对半平面列表的其余部分进行扫描，试图找出满足 $\eta_x(h_2) > 0$ 的某张半平面 h_2。如果能够找到这

样的一张半平面，h_1 和 h_2 就可以做为凭证，它们确保了一个唯一的字典序最小解的存在性。反过来，要是这样的 h_2 不存在，那么这个线性规划问题要么不可行，要么不存在一个字典序最小的解。可以将所有满足 $\vec{\eta}_x(h) = 0$ 的半平面构成一个一维线性规划问题，通过求解这个一维线性规划问题，就可以解决原来的问题。如果这个问题是可行的，就可以返回方向为(-1, 0)的一条射线ρ，该射线上的每个点都是可行的最优解。

至此，已经可以给出一个求解二维线性规划问题的通用算法如下：

算法 2DRANDOMIZEDLP(H, $\vec{c}$)

输入：线性规划问题(H, $\vec{c}$)，其中 H 为由 n 张半平面组成的一个集合，$\vec{c} \in \mathbb{R}^2$。

输出：若(H, $\vec{c}$)是无界的，则返回一条射线
若不可行，则返回两到三张作为凭证的半平面
否则，在使函数 $f_{\vec{c}}$ 达到最大的那些点中，返回字典序最小者。

1. 判断是否存在某个方向矢量 $\vec{d}$，满足 $\vec{d}\cdot\vec{c} > 0$，并且对所有的 $h \in H$ 都有 $\vec{d}\cdot\vec{\eta}(h) \geq 0$
2. **if** (这样的 $\vec{d}$ 存在)
3.   **then** 计算出 H'，并判断 H'是否可行
4.   **if** (H'是可行的)
5.   **then** 报告一条可以证明(H, $\vec{c}$)无界的射线，然后退出算法
6.   **else** 报告“(H, $\vec{c}$)是不可行的”，然后退出算法
7. 令 $h_1, h_2 \in H$ 为所谓的“凭证”半平面
    (* 它们确保了(H, $\vec{c}$)的有界性，同时说明按照字典序，该问题存在唯一的最小解 *)
8. 令 v_2 为 l_1 和 l_2 的交点
9. 设 $h_3, h_4, \ldots, h_n$ 为 H 中其余半平面的一个随机排列
10. **for** $i \leftarrow 3$ **to** n
11.   **do if** ($v_{i-1} \in h_i$)
12.   **then** $v_i \leftarrow v_{i-1}$
13.   **else** $v_i \leftarrow p$
            (* p 来自 l_i 上，它满足 H_{i-1} 中所有约束条件，并使函数 $f_{\vec{c}}$ 达到最大 *)
14.   **if** (p 不存在)
15.   **then** 令 h_j 和 h_k 为满足 $h_j \cap h_k \cap l_i = \emptyset$ 的两张凭证半平面
                    (* $j, k < i$，但是有可能 $h_j = h_k$ *)
16.   报告“该线性规划问题是不可行的”
                    提供凭证半平面 h_i、h_j 和 h_k
                    退出算法
17. **return** v_n

以上分析，可以归纳为如下定理：

〖定理 4.10〗

即使是在最坏情况下，使用不超过线性规模的空间，也可以在 $O(n)$的随机期望运行时间内，对包含 n 个约束条件的任何二维线性规划问题进行求解。

4.6 *高维空间中的线性规划

前面几节中所介绍的线性规划算法，可以推广至高维空间。当空间的维度不是很高时，与诸如单纯形法等传统的方法相比，我们更加倾向于采用这里介绍的算法。

在 $\mathbb{R}^d$ 中任取 n 张闭的半平面，组成集合 H。对于任意矢量 $\vec{c} = (c_1, \ldots, c_d)$，我们都找出一个点 $p = (p_1, \ldots, p_d) \in \mathbb{R}^d$，使得对于任一 $h \in H$，p 都在 h 之内，而且 p 使线性函数 $f_c(p) := c_1p_1 + \ldots + c_dp_d$ 达到最大。在该问题时有界的时候，为了保证解的唯一性，可以在所有使 $f_c(p)$达到最大的那些点中，按照字典序找出最小者，作为最优解顶点。

与平面的情况一样，我们也是递增式地逐一引入对应于各约束条件的半空间，并在此过程中不断对最优解进行更新。为了使之可行，在其中的每一步我们都同样需要确认，最优解存在而且唯一。为此，可以沿用上一节的做法——先判断该线性规划问题是否有界。如果有界，就可以找出一组共 d 个凭证半空间 $h_1, \ldots, h_d \in H$，它们确保了解的有界性，同时也确保了一个字典序最小解的存在性和唯一性。找出这些凭证半空间的具体方法将在稍后讨论，目前还是让我们将注意力放在主算法上。

在确认该线性规划问题的有界性之后，可以得到 d 个凭证半空间，令它们分别为 $h_1, \ldots, h_d$；至于 H 中其余的（n-d 个）半空间，则将它们随机打乱次序，并分别记作 $h_{d+1}, h_{d+2}, \ldots, h_n$。另外，我们将在引入前 i 个半空间后对应的可行解域记作 C_i，$d \le i \le n$。亦即

$$C_i := h_1 \cap h_2 \cap \ldots \cap h_i$$

将C_i中的最优解顶点（也就是使f_c达到最大的那个顶点）记作v_i。根据〖引理 4.5〗，可以得到一种在二维情况下维护和更新最优解顶点的简单方法——新的最优解顶点要么与此前的一样，要么必然落在最新引入的半平面h_i的边界线上。下面的这则引理，将这一结果推广到更高维的情况；其实，只要将〖引理 4.5〗的证明方法直接进行推广，就可以证明该引理。

〖引理 4.11〗

设 $1 \le i \le n$，而且 C_i 和 v_i 的定义如上。则有：

(i) 若 $v_{i-1} \in h_i$，则 $v_i = v_{i-1}$；

(ii) 若 $v_{i-1} \notin h_i$，则要么 $C_i = \emptyset$，要么 $v_i \in g_i$，其中 g_i 为 h_i 的边界超平面。

如果将 h_i 的边界超平面记作 g_i，那么只要找出交集 $g_i \cap C_{i-1}$ 中的最优解顶点，就可以找到 C_i 的最优解顶点 v_i。

然而，如何才能在 $g_i \cap C_{i-1}$ 中找出最优解顶点呢？在二维情况中，这可以在线性时间内轻而易举地办到——因为，这个问题的范围仅限于一条直线上。我们再来考察三维的情况。在三维空间中，g_i 为一张平面，故 $g_i \cap C_{i-1}$ 是一个二维的凸多边形区域。又该如何找到 $g_i \cap C_{i-1}$ 的最优解顶点呢？我们必须求解一个二维线性规划问题！原来定义于 $\mathbb{R}^3$ 中的线性函数 f_c，将在 g_i 中导出另一个线性函数；而我们的任务，就是在 $g_i \cap C_{i-1}$ 中找出一个点，使得这个新的函数达到最大。当然，要是 $\vec{c}$ 碰巧与 g_i 垂直，则 g_i 上的所有点都是一样“好”的（对该函数的取值相同）——在这种情况下，根据约定，应该取其中字典序的最小者。为此，可以选取一个适当的目标函数——例如，只要 g_i 与 x_1-坐标轴不垂直，就可以通过将矢量(-1, 0, 0)到 g_i 的投影当作矢量 $\vec{c}$。

因此，在三维情况中，可以按照如下方法来找出 $g_i \cap C_{i-1}$ 中的最优解顶点：首先计算出所有这 i-1 个半空间与 g_i 的交集，然后将下面四个矢量

$$\vec{c}, \begin{pmatrix}-1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\-1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\-1\end{pmatrix}$$

依次投影到 g_i 上，直到某一个的投影非零。这样，就得到了一个二维空间中的线性规划问题，我们可以利用 2DRandomiedLP 算法来解答它。

读到此处，读者或许会琢磨一个问题：如何才能处理 d-维空间中的一般性情况呢？在 d-维空间中，g_i 是一张超平面，即一个(d-1)-维子空间（subspace），而我们的任务，是要在交集 $C_{i-1} \cap g_i$ 中找出一个点，使得函数 f_c 达到最大。这是一个(d-1)-维的线性规划问题，因此，可以递归地调用我们的算法——当然，这是该算法针对(d-1)-维的一个版本。这种递归将一直进行下去，直到待解决的问题变成一个一维线性规划问题——这个问题可以在线性时间内直接求解。

我们依然需要判断，该线性规划问题是否有界；如果的确有界，还必须进一步找出合适的凭证。〖引理 4.9〗对任何维度都是成立的，我们首先要对此做一验证。实际上，该引理及其证明都不需做任何改动。根据这则引理，d-维的线性规划问题(H, $\vec{c}$)是有界的，当且仅当某个(d-1)-维的线性规划问题是不可行的。我们将通过递归调用，来解决这个(d-1)-维的线性规划问题。

如果这个(d-1)-维的线性规划问题是可行的，就可以得到一个方向矢量$\vec{d}$。而原先的那个d-维线性规划问题，要么沿着方向$\vec{d}$是无界的，要么就是不可行的。究竟是哪种情况呢？只要按照〖引理4.9〗中的定义找出H'，并检查(H', $\vec{c}$)是否可行，就可以判断出来。那么，这又应该如何判断呢？既然H'中每一个半空间的边界都与$\vec{d}$平行，这实质上就是另一个(d-1)-维的线性规划问题，因此只需再次进行递归调用，就可以解决这个问题。

反之，如果这个(d-1)-维的线性规划问题是不可行的，通过对它进行“求解”，就可以得到 k 个凭证半空间 $h_1, h_2, \ldots, h_k \in H$，其中 $k < d$——根据这些半空间，可以说明(H, $\vec{c}$)是有界的。若 $k < d$，则($\{h_1, \ldots, h_k\}$, $\vec{c}$)的最优解集必然是无界的。此时，这些最优解构成了一个(d-k)-维的子空间。我们将判断一下，限制于该子空间中的这个线性规划问题，关于字典序是否有界。如果不是，我们就返回这个解；否则，将不断重复这一过程，直到得到一组共 d 个凭证——此时，解必然是唯一存在的。

算法的主体框架如下。这里还是用 g_i 来表示半空间 h_i 的边界超平面。

算法 RANDOMIZEDLP(H, $\vec{c}$)

输入：线性规划问题(H, $\vec{c}$)，其中 H 为 $\mathbb{R}^d$ 中的一组共 n 个半空间，$\vec{c} \in \mathbb{R}^d$

输出：若(H, $\vec{c}$)无界，则返回一条射线
　　若该问题是不可行的，则返回不超过(d+1)个凭证半空间
　　否则，返回使函数 $f_{\vec{c}}(p)$达到最大的一个字典序最小点 p

1. 检查是否存在某个矢量$\vec{d}$，满足$\vec{d}\cdot\vec{c} > 0$，而且对所有 $h\in H$ 都有$\vec{d}\cdot\vec{\eta}(h) \geq 0$
2. **if** ($\vec{d}$存在)
3. 　**then** 计算 H'，并且检查 H'是否可行
4. 　　　**if** (H'是可行的)
5. 　　　　**then** 报告一条射线 (* 该射线的存在，说明(H, $\vec{c}$)是无界的 *)
　　　　　　退出算法
6. 　　　　**else** 报告“(H, $\vec{c}$)不可行”，同时还要提供凭证
　　　　　　退出算法
7. 令$\{h_1, h_2, \ldots, h_d\}$为证明“(H, $\vec{c}$)有界”的一组凭证
8. 令 v_d 为$\{g_1, g_2, \ldots, g_d\}$的公共交点
9. 生成 H 中其余各半空间$\{h_{d+1}, h_{d+2}, \ldots, h_n\}$的一个随机排列
10. **for** $i \leftarrow d+1$ **to** n
11. 　**do if** ($v_{i-1} \in h_i$)
12. 　　　**then** $v_i \leftarrow v_{i-1}$
13. 　　　**else** $v_i \leftarrow g_i$上的一个点 p

```
                    (* p 同时满足约束条件{h1, ..., hi-1}，并使函数 fc(p)达到最大 *)
14.                 if (p 不存在)
15.                     then 找出不超过 d 个凭证半空间，组成集合 H*
                             (* H*的存在，证明该(d-1)-维线性规划问题是不可行的 *)
16.                          报告"该线性规划问题"是不可行的，并
                             提供凭证集H* ∪ {hi}①;
                             退出算法
17.     return vn
```

RANDOMIZEDLP 算法的性能如何？下面的定理回答了这一问题。只要将 d 看作一个常数，就可以得出关于时间的一个 $O(n)$上界。尽管如此，不妨还是应该更加精细地分析一下 d 对运行时间的影响——参见下面这则定理证明的后面部分。

〖定理 4.12〗

对于每一固定的维数 d，任何含有 n 个约束条件的 d-维线性规划问题，都可以在 $O(n)$的期望运行时间内得到解答。

〖证明〗

我们必须证明的是：存在某个常数C_d，使得算法的期望运行时间不会超过C_dn。为此，可以对维数d进行归纳。首先，在二维空间中，根据〖定理 4.10〗可以直接得出这个结论，因此现在假定$d > 2$。这里的归纳过程，与 2-维情况的证明基本相同。

我们一开始所做的工作，是求解不超过 d 个(d-1)-维的线性规划问题。根据归纳假设，这需要花费 $O(dn) + dC_{d-1}n$ 的时间。

接下来，我们的算法还要花费 $O(d)$时间，以计算出 v_d。检查 $v_{i-1}\in h_i$ 是否成立，也要消耗 $O(d)$时间。因此，若暂时不计入第 13 行消耗的时间，则运行时间就是 $O(dn)$。

在第 13 行，需要将$\vec{c}$投影到 g_i 上（这需要 $O(d)$时间）；然后，分别将 i 个半空间与 g_i 求交（这需要 $O(di)$时间）。最后，还要递归地求解一个包含(i-1)个半空间的(d-1)-维线性规划问题。

定义一个随机变量 X_i 如下：如果 $v_{i-1}\notin h_i$，它就取值为 1，否则为 0。这样，该算法的总体期望运行时间就不会超过

$$O(dn) + dC_{d-1}n + \sum_{i=d+1}^{n}(O(di) + C_{d-1}(i-1)) \cdot E[X_i]$$

① 原书此处误作 $H^* \cup h_i$。——译者

为了界定 $E[X_i]$，我们将再次采用后向分析的方法。考虑已经加入了 $h_1, ..., h_i$ 之后的状态。既然此时的最优解顶点是 C_i 的一个顶点，它必然是由其中的 d 个半空间（通过相交）确定的。现在，将时间倒退一步。这样里所说的"倒退"，也可以理解为将这 i 个半空间之一删除。若删除之后最优解顶点会发生变化，则这个半空间必然是确定 v_i 的那 d 个半空间之一。因为 $h_{d+1}, ...,$ h_i 是随机排列的，所以这种情况发生的概率不会超过 $\frac{d}{i-d}$。

由此，关于该算法的期望运行时间，我们可以得出如下上界：

$$O(dn) + dC_{d-1}n + \sum_{i=d+1}^{n} (O(di) + C_{d-1}(i-1)) \cdot \frac{d}{i-d}$$

这个上界可以进一步放大至 $C_d n$，其中 $C_d = O(C_{d-1}d)$——也就是说，可以取一个与维数无关的常数 c，使得 $C_d = O(c^d \cdot d!)$。 ❑

若 d 的确是常数，则称"该算法在线性时间内运行结束"倒也无可厚非。但此结论极易被误解。因为随着 d 的提高"常"系数 C_d 将急速增长，故只有在维数不高的情况下，该算法才是有用的。

4.7 *最小包围圆

上面所介绍的随机技术，其实际的威力之大令人吃惊。除了常规的线性规划问题之外，这种方法还可以应用于其它各种各样的优化问题。以下就讨论这样的一个问题。

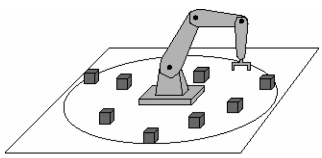


图4-23 机械手

如 图 4-23 所示，考察固定在工作平台上的一只机械手（robot arm）。其任务是：捡起散落在不同位置的多个零件，并移送到别的地方。那么，这只机械手的底座应该选在哪里呢？根据直觉，应选在机械手需够着的那些位置的"中心"。准确地讲，也就是包围这些点的那个最小圆的圆心——该位置的好处是，可使机械手的底座到它需要够着的那些点的最大距离最小化。于是可得如下问题：给定由平面上n个点所组成的一个集合P（对应于机械手需要够着的工作平台上的那些位置），试找出P的最小包围圆（smallest enclosing disc）——亦即，包含P中所有点、半径最小的那个圆。这个最小包围圆必然是唯一的——参见下面的〖引理 4.14〗，该引理将给出一个更具一般性的结论。

与此前各节的做法一样，这里也将为该问题设计一个随机增量式算法（randomized incremental algorithm）。首先，将 P 中各点打乱成一个随机排列 $p_1, \ldots, p_n$。令 $P_i := \{p_1, \ldots, p_i\}$。我们将逐一引入这些点，并在此过程中动态维护和更新 D_i——相对于 P_i 的最小包围圆。

线性规划问题中之所以能够方便地动态维护最优解顶点，是因为有一个很好的条件：每加入一张半平面，只要此前的最优解顶点包含于其中，就不必对其进行修改；否则，新的最优解顶点必然位于这个新半空间的边界上。那么，最小包围圆问题是否也具有类似性质呢？答案是肯定的。

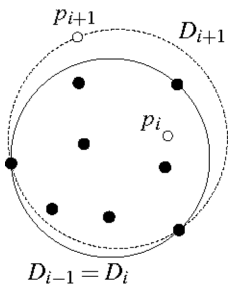


图4-24 若$p_i \in D_{i-1}$，则$D_i = D_{i-1}$

〖引理 4.13〗

设 $2 < i < n$，P_i和D_i的定义如上。则有：

(i) 若$p_i \in D_{i-1}$，则$D_i = D_{i-1}$（如 图 4-24 所示）；

(ii) 若$p_i \notin D_{i-1}$，则p_i必然落在D_i的边界上。

这个引理的证明将在稍后给出——因为我们首先想知道，应该如何利用这个性质，设计出一个类似于线性规划算法的随机增量式算法。

```
算法 MINIDISC(P)
输入：由平面上 n 个点组成的一个集合 P
输出：P 的最小包围圆
1.    将 P 中各点随机排列成 p1, ..., pn
2.    令 D2 为对应于{p1, p2}的最小包围圆
3.    for i ← 3 to n
4.        do if pi ∈ Di-1
5.              then Di ← Di-1
6.              else Di ← MINIDISCWITHPOINT({p1, ..., pi-1}, pi)
7.    return Dn
```

其中关键的一步，就是 $p_i \notin D_{i-1}$ 时的处理。我们需要另一个子程序，它能在“p_i 必须位于圆的边界上”这一限制性前提下，找到 P_i 的最小包围圆。那么，应该如何实现这个子程序呢？令 $q := p_i$。我们将再次采用几乎相同的算法结构——按照随机次序逐一引入 P_{i-1} 中的各点，在此过程中，还要

动态地维护和更新 $P_{i-1}\cup\{q\}$的最小包围圆。不同的是，这里多了一个附加的约束条件：这个最小包围圆的边界必须穿过 q。点 p_j 的引入，可以利用这样一个性质：若 p_j 包含于当前的最小包围圆中，则该最小包围圆无需改动；否则，p_j 必然落在新的最小包围圆的边界上。因此在后一种情况中，最小包围圆的边界必然同时穿过 q 和 p_j。我们可以得出如下的子程序：

```
算法 MINIDISCWITHPOINT(P, q)
输入：由平面上 n 个点构成的一个集合 P，以及另一个点 q
      (* 存在 P 的某个包围圆，其边界穿过 q *)
输出：在满足“边界穿过点 q”的前提下，P 的最小包围圆
1.  将 P 中各点打乱成一个随机排列：p1, …, pn
2.  令D1 为边界同时穿过q和p1 的最小圆 ①
3.  for j ← 2 to n
4.      do if (pj ∈ Dj-1)
5.            then Dj ← Dj-1
6.            else Dj ← MINIDISCWith2Points({p1, …, pj-1}, pj, q)
7.  return Dn
```

那么，应该如何在“其边界必须同时穿过 q_1 和 q_2”的前提下，找到一个给定集合的最小包围圆呢？我们依然可以“故伎重演”。也就是说，按照随机的次序逐一引入其中各点，在此过程中动态维护最优的圆。在引入点 p_k 时，倘若它落在当前的最小包围圆内，则什么也不必做；反之，要是 p_k 落在圆外，则新最小包围圆的边界必然穿过该点。如果是后一种情况，这个圆就必须同时穿过三个点：q_1、q_2 和 p_k。也就是说，只可能是唯一的一个圆——其边界由 q_1、q_2 和 p_k 确定。其详细过程可以描述为如下子程序：

```
算法 MINIDISCWith2Points(P, q1, q2)
输入：由平面上 n 个点构成的一个集合 P，以及另外的两个点 q1 和 q2
      (* 存在 P 的某个包围圆，其边界同时穿过 q1 和 q2 *)
输出：在满足“边界同时穿过点 q1 和 q2”的前提下，P 的最小包围圆
1.  令 D0 为边界同时穿过点 q1 和 q2 的最小圆
2.  for k ← 1 to n
3.      do if (pk ∈ Dk-1)
4.            then Dk ← Dk-1
5.            else Dk ← 边界同时穿过点 q1、q2 和 pk 的圆
6.  return Dn
```

① 即以 p_1q 为直径的圆。——译者

这样，就得出了计算给定点集最小包围圆的一套完整算法。在分析其性能之前，必须首先证明其正确性——为此，需要对该算法所利用到的那些性质给出证明。例如，我们曾经利用到这样一个性质：在引入新的一个点时，若它位于当前最优圆的外部，则新的最优圆的边界必然会穿过该点。

〖引理 4.14〗

设 P 为平面上的一个点集；设 R 为另一个（允许为空的）点集，而且 $P \cap R = \emptyset$；设点 $p \in P$。则有下列命题成立：

(i) **如果存在某个圆覆盖了 P，而且其边界穿过 R 中的所有点，那么这样的圆中必然存在唯一的最小者，记作** md(P, R)。

(ii) **如果** $p \in md(P \setminus \{p\}, R)$，**那么** $md(P, R) = md(P \setminus \{p\}, R)$。

(iii) **如果** $p \notin md(P \setminus \{p\}, R)$，**那么** $md(P, R) = md(P \setminus \{p\}, R \cup \{p\})$。

〖证明〗

(i) 如图 4-25 所示，假设存在半径相同的两个不同的包围圆D_0和D_1，其圆心分别在x_0和x_1。显然，P中所有的点必然落在交集$D_0 \cap D_1$中。我们将按照下面的方法，构造出一系列连续的圆$\{D(\lambda) \mid 0 \le \lambda \le 1\}$。取$D_0$和$D_1$边界（即$\partial D_0$和$\partial D_1$）的一个交点z。$D(\lambda)$的圆心是点$x(\lambda) := (1-\lambda)x_0 + \lambda_{x1}$，其半径为$r(\lambda) := d(x(\lambda), z)$。对于任何满足 $0 \le \lambda \le 1$ 的λ，我们都有$D_0 \cap D_1 \subset D(\lambda)$，因此作为一个特例，$\lambda = \frac{1}{2}$时也成立。于是，鉴于$D_0$和$D_1$都分别覆盖了P中的所有点，故$D(\frac{1}{2})$也必然如此。此外，$\partial D(\frac{1}{2})$也必然经过$\partial D_0$和$\partial D_1$的每个交点。由于$R \subset \partial D_0 \cap \partial D_1$，故有$R \subset \partial D(\frac{1}{2})$。也就是说，$D(\frac{1}{2})$不仅是P的一个包围圆，而且其边界同样会穿过R中的所有点。然而，就半径而言，$D(\frac{1}{2})$要严格小于D_0和D_1。因此，一旦出现半径相等的两个圆，而且它们的边界都各自穿过R中的所有点，就说明肯定存在另外一个（半径）更小的包围圆，而且这个圆的边界同样会穿过R中的所有点。于是，最小包围圆md(P, R)必然是唯一的。

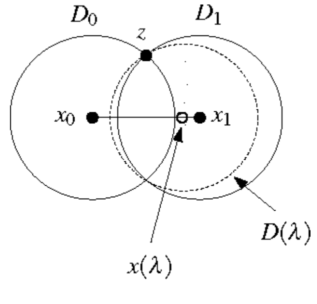


图4-25 覆盖P、其边界穿过R中所有点的圆必然唯一

(ii) 令 $D := md(P \setminus \{p\}, R)$。若 $p \in D$，则 D 必然包含 P，而且其边界穿过 R 中的所有点。不可能有任何更小的圆可以覆盖 P，而且其边界穿过 R 中的所有点——否则，这样的一个圆必

然是 P\{p}的一个包围圆，同时其边界穿过 R 中的所有点，这与 D 的定义矛盾。因此可以得出结论：D = md(P, R)。

(iii) 令 D_0 := md(P\{p}, R)，取 D_1 := md(P, R)。再次考虑前面所定义的一系列 D(λ)。我们注意到，D(0) = D_0，D(1) = D_1——实际上，由这一系列的圆，定义了从 D_0 到 D_1 的一个连续变形过程。根据假设，有 $p \notin D_0$；另外，还有 $p \in D_1$。因此，由其连续性，必然存在某个 $0 < \lambda^* \leq 1$，使得 p 正好落在 D(λ *)的边界上。与(i)的证明同理，可以得到 $P \subset D(\lambda^*)$和 $R \subset \partial D(\lambda^*)$。对任何 $0 < \lambda < 1$，D(λ)的半径必然会严格地小于 D_1 的半径。然而根据其定义，D_1 是 P 的最小包围圆，因此我们只有一种选择——$\lambda^* = 1$。也就是说，D_1 的边界必然穿过 p。 ❑

由〖引理 4.14〗可以得出一个推论：MINIDISC能够正确地计算出任何给定点集的最小包围圆。至于该算法的运行时间性能，在下面这则定理的证明中也给出了分析结果。

〖定理 4.15〗

平面上任意一组共 n 个点的最小包围圆，可以在 O(n)的期望运行时间内计算出来，而且为此需要的空间在最坏情况下不会超过线性规模。

〖证明〗

算法 MINIDISCWith2Points 中的每一轮迭代循环只需要常数时间，因此其运行时间为 O(n)；另外，它只需要线性的空间。至于算法 MINIDISCWITHPOINT 和 MINIDISC，它们也同样只需要线性的存储空间；因此，只需要对它们的期望运行时间做一分析。

对于算法 MINIDISCWITHPOINT，只要不计入其调用 MINIDISCWith2Point 的时间，其余计算所需时间为 O(n)。那么，需要进行这种调用的概率是多大呢？这里将再次采用后向分析的方法，来界定这一概率值。将子集{p_1, ..., p_i}固定，令 D_i 为覆盖{p_1, ..., p_i}、边界穿过 q 的最小圆。然后，假想着删去{p_1, ..., p_i}中的某个点。在哪些情况下，最小包围圆会发生变化呢？

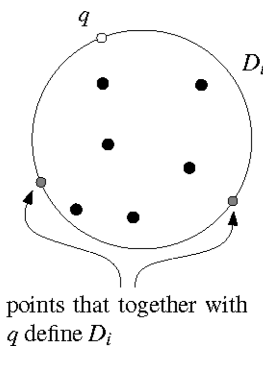


图4-26 除q外，至多还有两个点的删除会导致最小包围圆的缩小

只有当被删去的是原先落在圆周上的那三个点之一时，最小包围圆才有可能改变。因为 q 也是

落在该圆周上的诸点之一，所以能够导致最小包围圆缩小的最多只有两个点。p_i 是这样一个点的概率为$\frac{2}{i}$（如果该圆周同时穿过多于三个点，那么最小包围圆发生变化的可能性只会更小）。这样，就可以给出算法 MINIDISCWITHPOINT 期望运行时间的上界：

$$O(n) + \sum_{i=2}^{n} O(i) \cdot \frac{2}{i} = O(n)$$

接下来，同理可证，算法 MINIDISC 的期望运行时间也是 $O(n)$。 ❑

算法MINIDISC的改进余地很大。首先，在调用子函数MINIDISCWITHPOINT时，并不需要每次都生成一次随机排列。事实上，只须在算法MINIDISC的开头部分计算出一个这样的排列即可；此后对子函数MINIDISCWITHPOINT的每次调用，都可以沿用这一排列。此外，也不见得一定要分别写出三个独立的函数。事实上，完全可以写出一个统一的算法，来计算〖引理 4.14〗所定义的md(P, R)。

4.8 注释及评论

计算机辅助制造（computer-aided manufacturing）与几何计算密切相关，直到最近，计算几何界才开始加强对这类计算问题的研究。制造方法是个极为复杂的领域，这里仅仅是浅尝即止；Bose和Toussaint[72]就此做过详实的讨论。

“计算多张半平面公共交集”是一个古老的问题，人们对此做过充分的研究。正如我们将要在第 11 章中看到那样，从对偶的角度来看，这个问题等价于计算平面点集的凸包。这两个问题的研究历史都可以追溯到很早，Preparata和Shamos[323]已经为了我们列举了一些有关的解决方法。关于二维凸包的计算，本书第 1 章的“注释及评论”部分已做过更多的介绍。

“计算多个半空间公共交集”这一问题，在平面上以及三维空间中，都可以在$O(n\log n)$时间内解决。然而，随着维度的提高，这一问题的计算复杂度也会迅速提高。这是因为，在公共交集所对应的凸多胞体（convex polytope）中，（维数更低的）面的数目，可以多达$\theta(n^{\lfloor d/2 \rfloor})$[158]。因此，如果我们的目的只是找出一个可行点，就不应当去显式地计算出整个公共交集——因为，（随着维度的升高）这种方法将很快变得令人生厌。

线性规划是数值分析（numerical analysis）和组合优化（combinatorial optimization）领域的基本问题之一。要对这方面的文献做一综述，已经超出了本章所讨论的范围，因此我们只能将注意力集中于单纯形法及其变型 [139]、Khachiyan[234]以及Karmarkar[227]的多项式时间的解法。有关线性规划的更多介绍，可以参阅Chvatal[129]和Schrijver[339]等人的专著。

把线性规划作为计算几何领域的一个问题加以研究，始自Megiddo[273]。他通过分析证明：判断多个半空间的公共交集是否为空，要严格地比完整计算出这个交集更为容易。他给出了一个确定的线性规划算法，其运行时间可以表示为$O(C_d n)$的形式，其中C_d是一个只取决于维度的常数因子——这是第一个此类算法。在任何固定的维度中，该算法的复杂度均线性正比于n。然而，该算法的系数C_d却高达2^{2^d}。后来，这一结果被改进为3^{d^2} [130][153]。最近，又出现了若干随机算法 [132][346][354]，它们不仅更加简单而且更加实用。还有一些随机算法 [222][267]，其运行时间虽然是低于指数的，但关于维度仍不是多项式的。找出线性规划的一个强多项式算法（strongly polynomial algorithm）——也就是说，具有多项式级组合复杂度（combinatorial polynomial complexity）的算法——仍然是该领域中一个尚待解决的问题。

这里所介绍的随机增量式算法，是由Seidel[346]提出的，该算法非常简单，可以处理二维或更高维的问题。在对无界线性规划问题的处理上，本书介绍的方法有所不同，原算法采用的方法，是将参数M做符号式处理。或许，就其简洁性和效率而言，原算法的确要优于此处介绍的算法。之所以还是选择了这种介绍方式，目的在于说明d-维无界线性规划问题与(d-1)-维可行问题之间的联系。关于Seidel的那个版本，可以证明其复杂度因子C_d为$O(d!)$。

将这一算法推广到对最小包围圆问题的求解，要归功于Welzl[385]，他同时也说明了在高维空间中计算任何给定点集的最小包围球（smallest enclosing ball）、最小包围椭圆（smallest enclosing ellipse）以及最小包围椭球（smallest enclosing ellipsoid）的方法。Sharir和Welzl对这一方法做了进一步的推广，并提出了LP-类问题（LP-type problem）的概念——使用类似于这里所介绍的一种算法 [354][189]，这类问题都可以有效地得到解决。对于某些优化问题，在增加新的约束条件之后，问题的解要么不变，要么新的解必然部分地由这个新的约束条件定义，以至于问题的维度可以降低。一般而言，凡是具有这种性质的优化问题，都可以通过这种方法得到解决。也有人证明，LP-类问题的这些特殊性质，将会导出所谓的Helly-类定理（Helly-type theorem）。

借助于随机化（randomization）这一技术，往往能够得出简单而高效的算法。后面的各章还将陆续介绍更多这方面的例子。我们所需付出的代价是，算法的运行时间仅仅是一个期望的上界，而且正如我们已经看到的，在某些情况下算法的运行时间可能会更长。有些人揪住这一点不放，并据此说明，随机算法是不可靠的，因而不能采用。这种观点不无道理——试想，要是某家医院的特护中心或者某个核电站中的计算机正在使用一个随机算法，恐怕……。

然而反过来，确定性算法（deterministic algorithm）的完美性仅限于理论的意义。任何非平凡的算法都可能存在一些纰漏，纵然可以（从理论上）忽略这一点，但是在实践中，这类问题毕竟都有可能引起硬件的功能紊乱或所谓的“软错误”（soft error）——比如，由于环境中的α-辐射，核心内存中的某几个比特位发生了翻转。而随机算法通常更为简单，因而用更短的代码即可实现，这样，发生上述不测事故的概率也会更小。因此，随机算法不能及时给出正确答案的概率，未见得一定比

确定性算法出错的概率更高。另外，尽管随机算法某次具体运行的时间有可能超过我们估计的期望运行时间，但是只要选用一个足够大的常数因子，就可以将这种情况发生的概率降至任意低。

4.9 习题

习题 4.1 本章所讨论的铸模铸造问题，仅考虑了铸模只由一块组成的情况。这样，即使是球体这样简单的形状，也无法制造出来；当然，只要允许将铸模分为两块，就可行了。还有一些其它形状的物体，即使允许铸模由两块组成，依然不能制造出来；而使用某种由三块组成的铸模，则可以。你能举出这样的一个例子吗?

习题 4.2 试考虑平面上的如下铸造问题：给定一个多边形 $\mathcal{P}$ 以及相应的一个二维铸模，是否可以只通过单次平移运动，即将 $\mathcal{P}$ 从铸模中抽取出来呢？试给出一个线性时间的算法，对此做出判断。

习题 4.3 在三维的铸造问题里，假设在移出物体的过程中，我们不希望它紧贴着铸模的任何小平面滑动。做了这样的限制之后，我们原先导出的几何问题（即在一组半平面的公共交集中找出一个点），将会有何变化?

习题 4.4 任意给定一个可铸造的简单多面体（simple polyhedron）$\mathcal{P}$，设 f 为其顶面，$\vec{d}$为它的一个移出方向。试证明：与$\vec{d}$平行的任何直线与 $\mathcal{P}$ 相交，当且仅当与 f 相交。另外还请证明：与$\vec{d}$平行的任何直线 l 若与 $\mathcal{P}$ 相交，其交集 $l\cap\mathcal{P}$ 必然是连通的。

习题 4.5 任意给定包含 n 个顶点的一个简单多面体 $\mathcal{P}$。如果以 f 为顶面时 $\mathcal{P}$ 是可铸造的，那么一个必要的条件就是：与 f 毗邻的每一张小平面都完全处于 h_f（即 f 所在的那张平面）的同一侧。（当然，反过来并不一定成立——如果所有相邻的小平面都完全处于 h_f 的同一侧，那么即使以 f 为顶面，$\mathcal{P}$ 也并不见得一定是可铸造的。）试给出一个算法，在线性时间内，找出 $\mathcal{P}$ 所有满足这一条件的小平面。

习题 4.6* 试考虑增加如下限制条件之后的铸造问题：要求在将物体移出铸模时，只能沿着竖直（即与其顶面垂直的）方向平移。

a. 试证明：在这种情况下，可能作为顶面的候选只有常数个；

b. 试给出一个线性时间的算法，对任意给定的一个物体，判断是否存在符合上述限制条件的一个铸模。

习题 4.7 在将物体从铸模中取出时，可以不是要求通过单次平移，而是单次旋转（图 4-27）。为简单起见，我们只考虑铸造问题的平面版本，而且只限于顺时针方向的旋转。

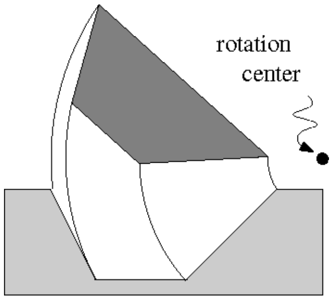


图4-27 通过旋转取出铸模

a. 试给出这样一个简单多边形（simple polygon）P 的实例：如果其顶面为 f，那么在只允许单次平移时，P 不是可铸造的；而在只允许围绕一个点旋转时，却是可铸造的。再举出另一个相反的例子：在只允许单次旋转时，P 不是可铸造的；而在只允许单次平移时，P 却是可铸造的。

b. 试证明："找到一个点，使得通过围绕该点的单次旋转，可以将 P 从铸模中取出"这一问题，也可以归约为"在一组半平面的公共交集中找出一个点"的问题。

习题 4.8 通过平面 z = 1，可以表示三维空间中任何 z-分量为正的矢量。那么，对于三维空间中 z-分量非负的那些矢量，又该如何表示呢？进而，我们又该如何来表示三维空间中的所有矢量呢？

习题 4.9 给定针对 n 个约束条件的任一三维线性规划问题，假设我们的目标是找出**所有的**最优解。试证明：解决该问题的任何一个算法，在最坏情况下的复杂度下界（lower bound）都是$\Omega(n\log n)$。

习题 4.10 设 H 为一组半平面，其中至少包含三张半平面，所有半平面的公共交集非空，而且它们的边界线不会全部平行。。如果某张半平面 $h \in H$ 没有为$\cap H$ 贡献任何边，我们就称之为"冗余的"（redundant）。试证明：对任何冗余的半平面 $h \in H$，必然存在另外的两张半平面 $h', h'' \in H$，使得 $h' \cap h'' \subset h$。试给出一个 $O(n\log n)$的算法，找出所有的冗余半平面。

习题 4.11 试给出满足如下要求的一个二维线性规划问题的实例：虽然该问题是有界的，却不存在字典序最小的解。

习题 4.12 试证明 RANDOMPERMUTATION(A)算法的正确性（亦即证明：A 的任何一种可能的排列，都有均等的机会成为该算法的输出）。另请证明：如果将其中第 2 行的 k 换作 n，该算法就不再正确了（亦即，这样将不能按照均等的概率生成各个排列）。

习题 4.13 本章给出了一个生成随机排列的线性时间算法。该算法需要借助于一个随机数发生器，从而在常数时间内生成一个介于 1 到 n 之间的随机整数。现在，假设我们能够使用的只是一个受限的随机数发生器，它能够在常数时间内给出一个随机的比特位（0 或者 1）。基于这样一个受限的随机数发生器，又该如何来生成一个随机排列呢？你

所给出的算法的时间复杂度是多少？

习题 4.14 下面这个假想式的算法，试图从包含 n 个实数的集合 A 中找出最大元素。

```
算法 PARANOIDMAXIMUM(A)
1.  if (card(A) = 1)
2.     then return A 中唯一的那个元素 x
3.     else 随机地从 A 中找出一个元素 x
4.          x' ← PARANOIDMAXIMUM(A\{x})
5.          if (x ≤ x')
6.             then return x'
7.             else (* 此时我们猜想 x 是最大元素 *)
                    不过为了万无一失，还得
                    将 x 与 A 中其它的card(A)-1 个元素逐一比较
8.                  return x
```

在最坏情况下，这个算法的运行时间是多少？（就第 3 行的随机选择而言，）其期望运行时间又是多少？

习题 4.15 简单多边形 $\mathcal{P}$ 被称作星形多边形（star-shaped polygon），如果其中存在某个点 q，使得对于该多边形内的任何点 p，线段$\overline{pq}$完全落在 $\mathcal{P}$ 内。给出一个算法，判断任何给定的简单多边形是否为一个星形多边形。该算法的期望运行时间必须是线性的。

习题 4.16 在 n 条平行的轨道上，n 列火车正在匀速行驶，其速度分别 $v_1, v_2, \ldots, v_n$。在 t = 0 时刻，各列火车的位置分别处于 $k_1, k_2, \ldots, k_n$。试给出一个 $O(n\log n)$的算法，找出可能在某一时刻处于领先位置的所有火车。（为此，你可能需要用到我们用以计算一组半平面公共交集的算法。）

习题 4.17* 只需借助于一个计算（如〖引理 4.14〗所定义的）md(P, R)的子程序 MINIDISCWITHPOINTS(P, R)，即可实现MINIDISC算法。试描述具体的实现方法。在你的算法中，只允许在所有计算开始之初生成一次随机排列，而其它时候都不允许再次生成随机排列。

5

正交区域查找：数据库查询

乍看起来，数据库与几何风马牛不相及。然而实际上，针对数据库中数据的许多类型的分析——从此我们称之为查询（query）——都可以从几何的角度来加以理解。为此，我们将数据库中的每条记录对应到多维空间中的一个点，从而将针对记录的查询转换为针对点集的查询。以下通过一个例子来加以说明。

试考察一个人事管理的数据库。这样一个数据库所存储的信息，包括每位员工的姓名、住址、生日和工资等等。对该数据库的一次典型的查询，可能是要求报告出所有出生于 1950 年至 1955 年

之间、月收入在$3,000 至$4,000 之间的所有员工。为将此描述为一个几何问题，我们将每位员工表示为平面上的一个点：点的第一个坐标是出生日期，比如可以表示为一个整数，其值为 10,000 × 年 + 100 × 月 + 日；第二个坐标是月收入。在每个点处，我们还可以同时存储该员工的其它信息，比如姓名与住址。这样，对数据库中“出生于 1950 年至 1955 年之间、月收入在$3,000 至$4,000 之间的所有员工”的查询，就转化为一个几何查询：报告出第一个坐标介于 19,500,00 和 19,550,000 之间、第二个坐标介于 3,000 到 4,000 之间的所有点。换而言之，给定一个与坐标轴平行的查询矩形，我们希望找出落在其中的所有点（参见 图 5-1）。

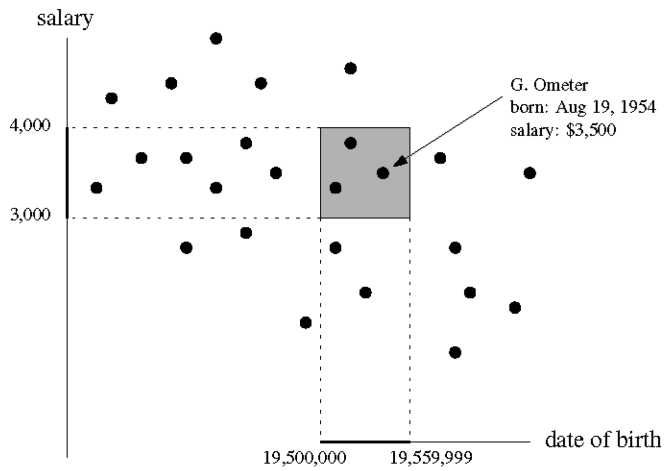


图5-1 从几何的角度来理解对数据库的查询

要是我们同时还掌握每位员工家中孩子的数目，并希望能够进行类似“报告出生于 1950 年至 1955 年之间、月收入在$3,000 至$4,000 之间并且有两到四个孩子的所有员工”的查询，情况又将如何？在这种情况下，我们可以将每位员工表示为三维空间中的一个点：点的第一个坐标是出生日期；第二个坐标是月收入；而第三个坐标则是孩子的数目。

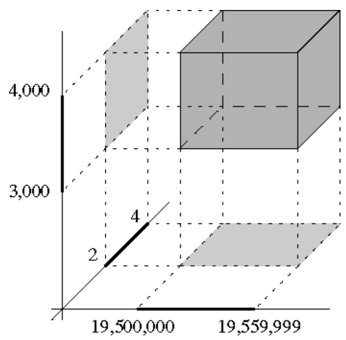


图5-2 将数据库查询转化为空间的区域查找

于是如 图 5-2 所示，为了回答上述查询，要做的就是在给定一个与坐标轴平行的长方体 [19,500,000 : 19,550,000] × [3,000 : 4,000] × [2 : 4]之后，报告出落在其中的所有点。一般而言，如果数据库中的每个记录包含d个数据域，那么为回答此类查询，可以将所有记录转化为d维空间中的一

组点。这样，查询各数据域分别介于特定范围内的记录，就转化为“报告出落在某个与坐标轴平行的d维（超）长方体内的所有点”的问题。在计算几何（computational geometry）中，这类查询被称为矩形区域查询（rectangular range query），或者正交区域查找（orthogonal range query）。本章将研究支持这类查找的数据结构。

5.1 一维区域查找

在着手处理二维或更高维矩形区域查找问题（range searching problem）之前，让我们首先来看看一维的情形。给我们的输入数据是一维空间中——即一条直线上——的一个点集。需要查询的，是该点集中落在某个一维矩形——即某个区间[x : x']——内的所有点。

设直线上给定的这一点集为 $P := \{ p_1, \ldots, p_n \}$。这个一维的区域查找问题可以有效的解决，为此可以利用某种众所周知的数据结构——平衡二分查找树 $\mathcal{T}$。（当然，直接使用数组也可以圆满地解决这一问题。然而，这种方法既不能推广到更高维的空间，也不能有效地支持对 P 的动态更新。）$\mathcal{T}$ 的叶子分别存储了 P 中的各点，而 $\mathcal{T}$ 的内部节点则记录了划分的数值，用来引导查找。将（内部）节点 v 所对应的划分值记为 x_v。我们假设：v 的左子树中存储了（坐标）不超过 x_v 的所有点，而其右子树中则存储了（坐标）严格大于 x_v 的所有点。

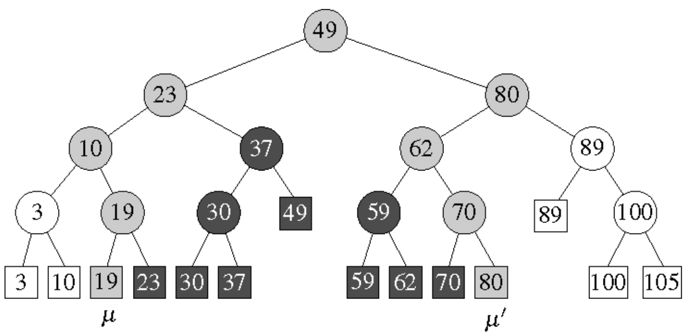


图5-3　在二分查找树上的一维区域查找

为了报告出落在待查询区域[x : x']内的所有点，可以如下进行。在$\mathcal{T}$中分别查找x和x'，设两次查找分别终止于叶子μ和μ'。于是，位于区间[x : x']之内的点，就对应于介于μ和μ'之间的那些叶子（可能还要加上μ或μ'本身）。以如 图 5-3 所示的树为例，如果查询的区间是[18 : 77]，需要报告的就是深灰色的那些叶子，再加上叶子μ自己。那么，如何才能找出介于μ与μ'之间的那些叶子呢？由 图 5-3 可以看出：在对应于μ和μ'的两条查找路径之间，存在一些子树；而需要找出的那些叶子，都分别来自于其中的某棵子树。（在 图 5-3 中，这些子树已被标为深灰色，而分布于这两条查找路径上的各个顶点都是淡灰色的。）更准确地讲，我们所选出的每一棵子树，都以介于这两条查找路径之间的某个节点v为根，而且v的父亲节点位于某条查找路径之上。为了找到这类节点，首先要确定与x和x'对应的两条查找路径开始分叉的位置，记这个节点为v_{split}。这可以由下面的子程序来完成。分别将节

点v的左、右孩子记为lc(v)和rc(v)。

```
算法 FINDSPLITNODE(T, x, x')
输入：树T，以及两个数值 x 和 x'，x ≤ x'
输出：从树根出发、分别通往 x 和 x'的两条路径的分叉点 v
        (* 若这两条路径完全重合，则返回它们共同的终点 *)
1.   v ← root(T)
2.   while (v 还不是叶子) 且 (x' ≤ x_v 或 x > x_v)
3.      do if (x' ≤ x_v)
4.            then v ← lc(v)
5.            else v ← rc(v)
6.   return v
```

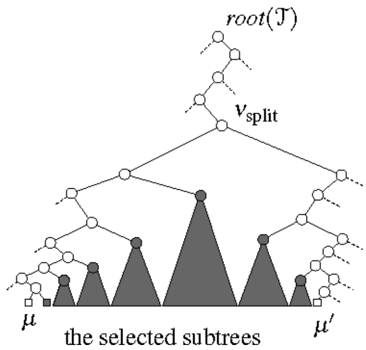


图5-4 从v_{split}出发分两路前进

现在，从v_{split}出发，沿着x对应的查找路径前进。如 图 5-4 所示，每向左前进一步，就枚举出该处右子树中的所有叶子——因为，这棵子树必然介于上述两条查找路径之间。类似地，再从v_{split}出发沿着x'对应的查找路径前进，并且每向右前进一步，都枚举出该处左子树中的所有叶子。最后，别忘了还要分别检查一下两条查找路径终点处的叶子，因为它们对应的点可能位于区间[x : x']之内，也可能位于其外。

下面对查找算法做一详细描述。这里使用一个名为 REPORTSUBTREE 的子程序：给定以某个节点为根的一棵子树，该子程序通过遍历（traversal），报告出所有叶子对应的点。鉴于任何二叉树中的内部节点都少于其中的叶子，故该子程序的运行时间将线性正比于被报告出来的点数。

```
算法 1DRANGEQUERY (T, [x : x'])
输入：二分查找树T，待查询区域[x : x']
输出：存储于T中、落在[x : x']内的所有点
1.   v_split ← FINDSPLITNODE(T, x, x')
2.   if (v_split 是叶子)
```

```
3.          then 检查 v_split 对应的点是否应该被报告
4.          else (* 沿着通往 x 的路径，报告路径右侧每一子树内的所有点 *)
5.                v ← lc(v_split)
6.                while (v 还不是叶子)
7.                   do if (x ≤ x_v)
8.                         then REPORTSUBTREE(rc(v))
9.                               v ← lc(v)
10.                        else v ← rc(v)
11.               检查叶子 v 对应的点是否应该被报告
12.               类似地，沿着通往 x'的路径，报告路径左侧每一子树内的所有点
                        检查该路径终点（叶子）对应的点是否应该被报告
```

下面首先证明该算法的正确性。

〖引理 5.1〗

算法 1DRANGEQUERY 所报告出来的，恰好就是落在查找区间内的所有点。

〖证明〗

首先证明，被报告出来的每个点p都落在查找区间之内。如果p对应的叶子是x或x'所对应路径的终点，那么p将经过明确的测试，被确定是否位于查找区间之内。否则，p必然是在对REPORTSUBTREE的某次调用中被报告出来的。（不失一般性地）假设这次调用发生在沿着通往x的路径前进的过程中。设v为该路径上的一个节点，而且p就是通过调用REPORTSUBTREE(rc(v))被报告出来的。因为v属于v_{split}的左子树（当然，rc(v)亦是如此），所以必有$p \le x_{v_{split}}$。通往x'的查找路径在v_{split}处朝右方前进，故有$p < x'$[①]。另一方面，通往x的查找路径在v处朝左方前进，而p却位于v的右子树内，因此有$x < p$。于是，就有 $p \in [x : x']$[②]。另一种情况（即p是在通往x'的途中被报告出来的），也可以完全对称地得到证明。

接下来需要证明：位于该区间之内的每一个点 p 都会被报告出来。设μ为点 p 所对应的叶子；在μ的所有祖先中，设 v 为被查询算法访问过的最低者。我们声称：$v = \mu$—— 也就是说，p 必然会被报告出来。若非如此，即 $v \neq \mu$。注意到 v 不可能通过调用 REPORTSUBTREE 而被报告出来——否则，该节点的所有后代都会被访问到。因此，v 或者位于通往 x 的查找路径之上，或者位于通往 x'的查找路径之上，或者同时位于这两条路径之上。这三种情况大同小异，故只考虑最后一种。首先假设μ位于 v 的左子树中。于是通往 x 的查找路径在 v 处向右方前进（否则，v 就不可能是μ的祖先中被访问过的最低者）。然而这却意味着 $p < x$。类似地，如果μ位

① $p \le x_{split} < x'$。——译者

② 更准确地，应该是 $p \in (x : x')$。——译者

于 v 的右子树中，那么通往 x'的查找路径也将在 v 处向左方前进，于是有 x' < p。无论是那种情况，都与“p 位于区间内”相悖。 ❑

现在来考察数据结构的效率。这里采用的是平衡二分查找树，占用$O(n)$空间，并且可以在$O(n\log n)$时间内构造出来。查询时间呢？在最坏的情况下，所有的点都位于区间内。此时，查询时间是$\Theta(n)$，似乎不是很好。实际上，无需借助任何数据结构，就可以实现$\Theta(n)$的查询时间——只要逐一将各点与待查询区域进行比较即可。另一方面，在所有点都需要报告的时候，$\Theta(n)$的查询时间也是不可避免的。因此，我们需要对查询时间做更为精细的分析。这种精细的分析不仅要考虑到集合P所包含的点数n，还要考虑到需要报告出来的点数k。换而言之，我们将证明：这一查找算法是输出敏感的（output-sensitive）——在第 2 章中，我们已经遇到过这一概念。

你应该记得，每次 REPORTSUBTREE 调用所需的时间线性正比于其报告出来的点数。因此，此类调用所需的时间总共为 $O(k)$。其它被访问过的节点，都位于通往 x 或（和）x'的查找路径之上。鉴于 T 是平衡的，故这种路径的长度为 $O(\log n)$。对于其中的每一节点只需 $O(1)$时间，故花费于这类节点的总时间为 $O(\log n)$。这样，查询的时间就是 $O(\log n + k)$。

关于一维区域查找的上述结果，可以总结为如下定理：

〖定理 5.2〗

给定由一维空间中任意 n 个点构成的集合 P。可以使用 $O(n)$空间，在 $O(n\log n)$时间内构造一棵平衡二分查找树以存储 P。这样，可以在 $O(k + \log n)$时间内查找出任何区间内的所有点（其中，k 是实际被查找出来的点数）。

5.2 kd-树

现在来讨论二维矩形区域查找的问题。设 P 为由平面上任意 n 个点构成的集合。本节接下来的部分都假设 P 中任何两个点的 x 坐标都互异，y 坐标也是如此。这种限制与实际情况不甚相符——当这里的点表示的是员工，而坐标分别对应于工资或者孩子数目之类的数值时，尤其如此。然而幸运的是，利用某种技巧，这些限制都是可以消除的。第 5.5 节将介绍这一技巧。

如 图 5-5，所谓针对P的一次二维矩形区域查找，就是要从P中找出落在某一待查询矩形$[x : x'] \times [y : y']$之内的所有点。点$p := (p_x, p_y)$落在该矩形之内，当且仅当

$$p_x \in [x : x'] \text{ 而且 } p_y \in [y : y']$$

由此我们可以认为，每次二维矩形区域查找，都可以分解为两次一维子查找——其中一次沿 x 方向

进行，另一次沿 y 方向进行。

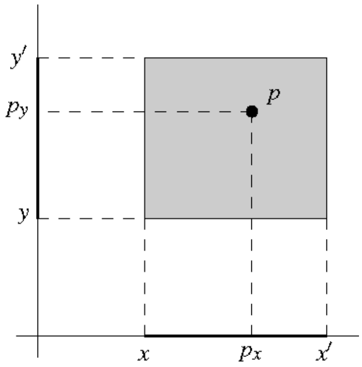


图5-5 二维矩形区域查找

在前一节中，我们见过一种支持一维区域查找的数据结构。那么。应该如何对那种结构——实际上，它不过就是一棵二分查找树——做一般化推广，以支持二维区域查找呢？我们先从递归的角度来看看二分查找树的定义：将（一维）点集接近平均地分为两个子集；其中一个子集，由所有不超过某一门槛值的那些点构成，而另一个子集则由超过某一门槛值的那些点构成。门槛值存储于根节点处，而上述两个子集将分别递归地存储于这两棵子树之中。

在二维情况下，每个点都拥有两个基本的数值：其x坐标和y坐标。因此，首先沿x坐标方向做一次划分，然而沿y方向做一次，接着再次沿x方向划分，如此下去。更准确地说，这一过程可以定义如下。首先，如 图 5-6 所示，在根节点处，通过一条垂线l将集合P划分为大小接近的（左、右）两个子集。

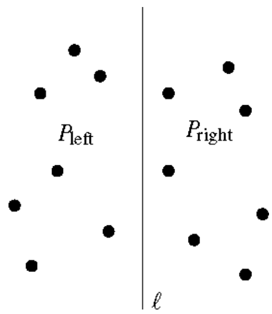


图5-6 沿垂线l将集合P划分为规模接近的两个子集

该分割线存储于根节点处。位于分割线左侧的那个子集记作P_{left}，它被存储于左子树中；位于分割线右侧的那个子集记作P_{right}，它被存储于右子树中。在根节点的左孩子处，继续通过一条水平线，将P_{left}划分为（上、下）两个子集：位于该分割线以下或者落于其上的那些点，被存储于该左孩子的左子树中；而位于分割线以上的那些点，则被存储于右子树中。该左孩子将记录下水平分割线的位置。类似地，P_{right}也将被某条水平线划分成两个子集，它们分别被存储于右孩子的左、右子树中。对于根节点的每个孙子，再次通过一条垂线进行划分。一般地，对于深度为偶数的节点，使用垂线

进行划分；对于深度为奇数的节点，将使用水平线进行划分。图 5-7 显示了这种划分的进行过程，以及对应的二分查找树的结构。这样的一棵树称作kd-树（kd-tree）。最初，这个名字的含义为"k 维树"（k-dimensional tree）。比如，以上描述的就是一棵 2d-树。不过时至今日，其最初的含义已经不复存在，现在都将 2d-树称为"2 维kd-树"（2-dimensional kd-tree）。

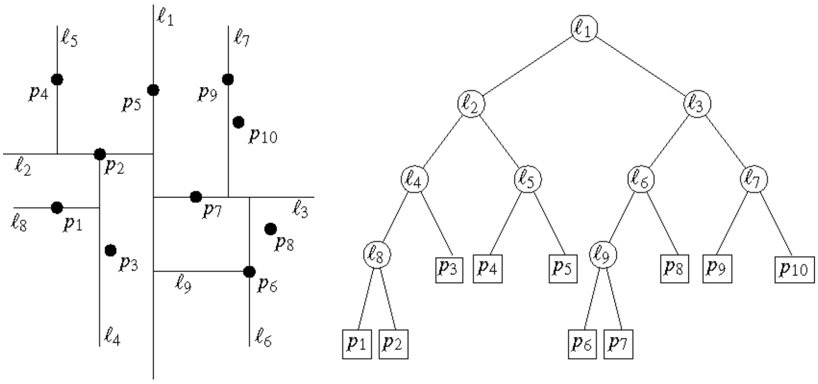


图5-7 一棵kd-树：左侧所示的是对整个平面的一个划分，右侧为其对应的二分查找树

可以使用如下子程序来构造一棵 kd-树。该子程序有两个输入参数：一个点集以及一个（非负）整数。第一个参数就是我们需要为之建立 kd-树的点集；初次调用时，它就是集合 P 本身。第二个参数为递归深度，即在递归调用构造子程序时，对应子树根节点的深度。初次调用时，后一参数设置为零。深度在此很重要——因为正如上面所解释的，这决定了我们究竟是使用垂线还是水平线来进行划分。该子程序最后将返回（被构造出来的）kd-树的根节点。

```
算法 BUILDKDTREE (P, depth)
输入：点集 P 以及当前的深度 depth
输出：与 P 对应的 kd-树的根节点
1.    if (P 只包含一个点)
2.       then return (存储该点的叶子)
3.       else
            if (depth 为偶数)
4.             then 沿着通过 P 内各点 x-坐标中值的垂线 l，将 P 划分为左、右两个子集
                    (* 记 P1 为对应于位于 l 左侧或者落在 l 上诸点的子集 *)
                    (* 记 P2 为对应于位于 l 右侧诸点的子集 *)
5.             else 沿着通过 P 内各点 y-坐标中值的水平线 l，将 P 划分为上、下两个子集
                    (* 记 P1 为对应于位于 l 下方或者落在 l 上诸点的子集 *)
                    (* 记 P2 为对应于位于 l 上方诸点的子集 *)
6.    v_left ← BUILDKDTREE (P1, depth+1)
```

7.	$v_{right} \leftarrow$ BUILDKDTREE (P_2, depth+1)
8.	生成一个节点 v 以存储直线 l，并分别将 v_{left} 和 v_{right} 置成 v 的左、右孩子
9.	**return** v

此算法遵循这样一个约定：恰好被分割线穿过的点——也就是点集内 x-坐标或者 y-坐标的中值（median）——将被归入分割线左侧或者下方的子集。为了使这种约定行之有效，需要相应地将 n 个数的中值定义为“从小到大排列的第$\lfloor \frac{n}{2} \rfloor$数”。也就是说，如果只有两个数，中值就是其中更小的那个——唯此才能保证算法的正常运行。

在开始讨论查找算法之前，我们首先来确定构造一棵二维 kd-树所需要的时间。在每次递归调用中，最耗时的步骤就是确定分割线的位置。根据递归深度的奇偶不同，为此需要确定 x-坐标或者 y-坐标的中值。可以在线性时间内找到中值。然而，这种线性时间的中值查找算法相当复杂。这里更好的一种办法是，预先对所有点按照 x-坐标和 y-坐标分别进行一次排序，这样，传递给上述子函数的参数 P 将由两个列表组成，它们分别记录了 x-坐标或 y-坐标的排序结果。有了这两个列表，就很容易在线性时间内找到（当深度为偶数时的）x-坐标中值或者（当深度为奇数时的）y-坐标中值。而且，根据给定的列表，也可以很容易地在线性时间内构造出两个子列表，供（下一层的）两次递归调用使用。于是，构造过程所需的时间 T(n)符合下列递推关系：

$$T(n) = \begin{cases} O(1) & \text{如果} n = 1 \\ O(n) + 2T(\lceil \frac{n}{2} \rceil) & \text{如果} n > 1 \end{cases}$$

该递归式的解为 $O(n\log n)$。另外，消耗在（对 x-和 y-坐标）预排序方面的时间也未超过这一上界（upper bound）。

为了确定存储空间的上界，请注意到：kd-树中的每匹叶子都存储了 P 中一个点，而不同叶子所存的点也不同。因此，共有 n 匹叶子。另外，kd-树是二叉树，而且每个（无论是叶子还是内部）节点都占用 $O(1)$空间，故总的存储量为 $O(n)$。这样就得到了如下引理：

〖引理 5.3〗

给定由任意n个点组成的一个集合[①]，其对应的kd-树占用$O(n)$空间，并可以在$O(n\log n)$时间内构造出来。

现在来讨论查找算法。根节点所存储的分割线，将整个平面划分为（左、右）两张半平面（half-plane）。左半平面内的点存储于左子树中，右半平面内的点则存储于右子树中。就此意义而言，根节点的左、右子树分别对应于左、右半平面。（按照此前BUILDKDTREE算法的约定，正好落

① 作者这里直接将结果从二维点集推广至任意维情况。幸运的是，这一结果是正确的。——译者

在分割线上点应归入左子集，故在这里，左半平面的右边界是闭的，而右半平面的左边界则是开的。）kd-树中的其它节点，也分别对应于平面的某个子区域。例如根节点的左子树的左子树所对应的子区域，其右边界是由根节点所对应的分割线界定的，而其上边界则是由根节点左子树所对应的分割线界定的。一般而言，任一节点v所对应的子区域是一个矩形，不过，这个矩形在某些边的方向上可能是无界的（unbounded）。这个矩形之所以在某些方向是有界的，是由于受到了v的祖先们所对应的分割线的界定（参见 图 5-8）。

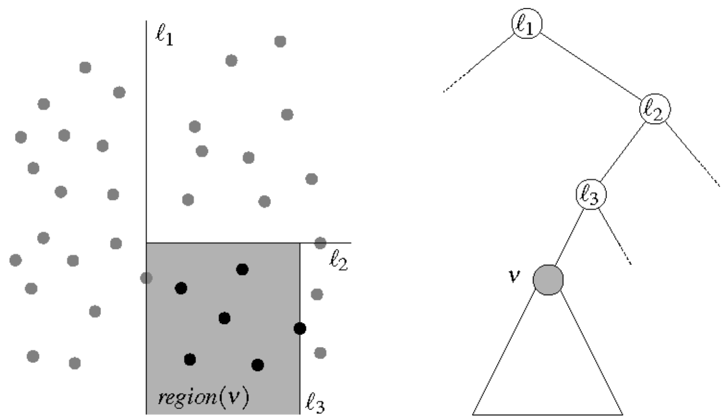


图5-8 kd-树中各节点与平面上某一子区域的对应关系

我们将对应于节点v的子区域记为region(v)。显然，对应于kd-树根节点的区域就是整个平面。请注意：一个点存储于以v为根的某棵子树之中，当且仅当它落在region(v)之内。例如在 图 5-8 中，以v为根的子树所存储的，就是黑色的点。因此，只有当region(v)与待查询区域相交时，我们才有必要对以v为根的子树进行查找。根据这一观察结果，可以得到如下查找算法：对kd-树做遍历（traversal）——不过，只访问那些其对应子区域与查找矩形相交的节点。如果遇到某个其区域完全包含于查找矩形之中，就将其中的点 [①]报告出来。要是遇到一匹叶子，就需要专门检测一下，以确定其对应的点是否落在待查询区域之内；如果是，也将它报告出来。图 5-9 显示了算法过程。（请注意：图 5-9 所示的kd-树并不是由BUILDKDTREE算法构造出来的——在这里，并不总是按照中值来进行划分。）如果待查询区域为其中的灰色矩形，那么其中灰色的那些节点将会被访问到。那个标有星号的节点，对应于一个完全包含于待查询区域之中的子区域（在图中，这一子区域被标记为更深的颜色）。于是，在对以此节点为根的（深灰色）子树遍历之后，存储于其中的所有点都将被报告出来。其它被访问到的叶子，分别对应于与待查询矩形局部相交的某一子区域。于是，需要对它们所对应的点进行测试，以确定其是否落在待查询区域之内——在此处的例子中，p_6和p_{11}将会以这种方式被报告出来，而p_3、p_{12}和p_{13}则不然。整个查找算法可以描述为下面的递归式程序，该程序需要两个参数——kd-树的根节点v，以及待查询区域R。它通过调用一个子程序REPORTSUBTREE(v)，对以v为根的子树进行遍历，以报告出其中所有的叶子。请记住我们的约定：lc(v)和rc(v)分别表示节点v的左、右孩子。

① 亦即，对应子树的所有叶子。——译者

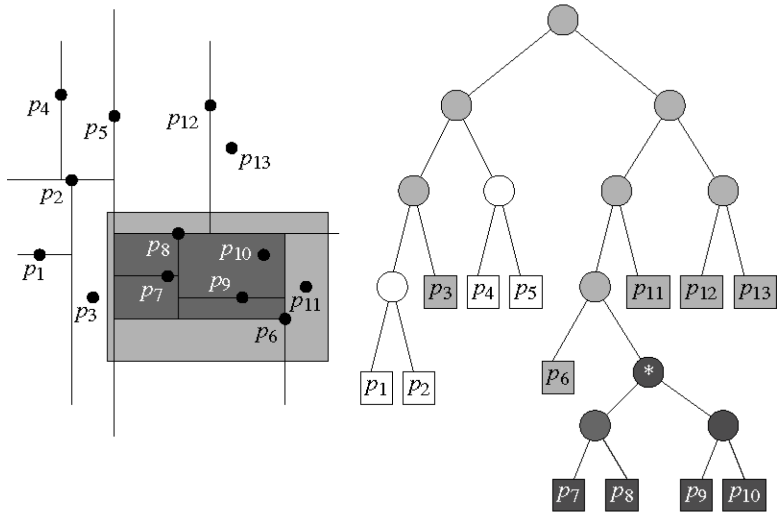


图5-9 对kd-树的查找

算法 SEARCHKDTREE(v, R)

输入：kd-树的根节点 v，以及待查询区域 R

输出：所有以 v 为祖先、位于 R 之内的叶子所对应的点

1. **if** (v 是叶子)
2. **then** (要是 v 落在 R 之内，则把它报告出来)
3. **else** **if** (region(lc(v))完全包含于 R 之中)
4. **then** ReportSuntree(lc(v))
5. **else** **if** (region(lc(v))与 R 相交)
6. **then** SEARCHKDTREE(lc(v), R)
7. **if** (region(rc(v))完全包含于 R 之中)
8. **then** ReportSuntree(rc(v))
9. **else** **if** (region(rc(v))与 R 相交)
10. **then** SEARCHKDTREE(rc(v), R)

该查询算法主要进行的测试，是判断待查询区域R是否与某一节点v所对应的子区域相交（如 图5-10 所示）。为了进行这种测试，一种直截了当的方法就是：通过预处理，计算出对应于每一节点v的region(v)，然后将其存储下来。然而，实际上并不需要这样做。在递归调用的过程中，可以借助于各内部节点所对应的分割线，维护对当前子区域的描述。例如，给定处于偶数深度的某个节点v所对应的子区域region(v)，其左孩子所对应的子区域可以这样获得：

$$\text{region}(lc(v)) = \text{region}(v) \cap l(v)^{\text{left}}$$

其中，l(v)就是存储于 v 处的分割线，而 $l(v)^{\text{left}}$ 则是 l(v)左侧（包含 l(v)）的半平面。

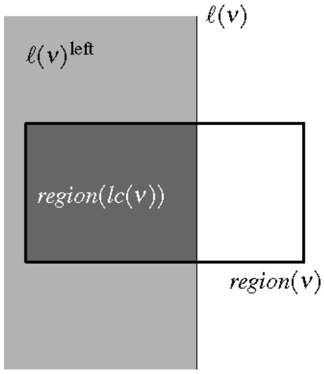


图5-10 判断R是否与region(v)相交

请注意：上述查找算法从未假设待查询区域必须是矩形。实际上，对于其它形状的待查询区域，这一算法同样适用。

下面对每次矩形区域查询所需的时间做一分析。

〖引理 5.4〗

如果待查询区域为与坐标轴平行的矩形，那么对于存储了任意 n 个点的一棵 kd-树，每次查询都可以在 $O(\sqrt{n}+k)$时间内完成，其中 k 为实际被报告出来的点数。

〖证明〗

首先请注意以下事实：对一棵（子）树进行遍历，并报告其中每匹叶子所存储的点，需要的时间将线性正比于被报告出来的点数。因此，第 4 行与第 8 行对所有子树进行遍历所需的总时间为$O(k)$，其中k为实际被报告出来的点数。因此，下面只需估计出那些被访问过、但不是在对某棵子树进行遍历的过程中被访问过的那些节点的数目。这些点也就是 图 5-9 中那些淡灰色的节点，每个这类节点v所对应的子区域region(v)，都会与待查询区域真相交（properly intersecting）——即region(v)与之相交，但并非完全包含于其中。换而言之，待查询区域的边界必然与region(v)相交。为了估计出这类节点的数目，只需估计一条垂线至多可能与多少个子区域相交。这样，就可以估计出与待查询矩形左边界、右边界可能相交的子区域数目的上界。按照同样的方法，也可以估计出与待查询矩形上边界、下边界可能相交的子区域数目的上界。

给定一棵kd-树$\mathcal{T}$，任取一条垂线l。记该kd-树的根节点所对应的分割线为l(root($\mathcal{T}$))。直线l或者与l(root($\mathcal{T}$))左侧区域相交，或者与l(root($\mathcal{T}$))右侧区域相交，然而，绝不可能与二者同时相交。对于所有由n个点构成的kd-树，记其中可能与l相交的子区域的（最大）数目为Q(n)。根据以上的观察结果，Q(n)似乎应该满足这样的递推关系：$Q(n) = 1 + Q(n/2)$。然而，这却是不对的——因为，在根节点的孩子们那里，分割线都是水平的。这就意味着，若直线l与

region(lc(root(*T*)))[①]相交，则它必然与lc(root(T))的两个孩子所对应的子区域同时相交。于是，我们接下来面临的递归条件与最初的情况有所不同，故上面的递推关系式并不成立。为了解决这个问题必须设法保证，递归条件依然与此前相同——子树的根节点也必须对应于一条垂直的分割线。于是，我们就要将Q(n)定义为：在根节点对应于一条垂直分割线、存储了n个点的kd-树中，l与其中子区域相交的最大数目。为了得到Q(n)的递推关系，现在必须从树根下推两层（如图 5-11 所示）。

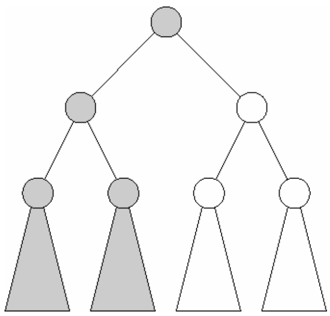


图5-11 每次向下递推两层

在这棵 kd-树中，深度为 2 的四个节点分别对应于含$\frac{n}{4}$个点的某个子区域。（更精确地说，其中任何一个子区域最多含有$\lceil\lceil\frac{n}{2}\rceil/2\rceil = \lceil\frac{n}{4}\rceil$个点。然而从渐进分析的角度来看，这点误差还不足以影响到下列递推式的成立。）这四个节点中有两个所对应的子区域与 l 相交，故只需递归地去估计这两个节点所对应的子树中与 l 相交的子区域数目。此外，l 还与根节点所对应的子区域，以及根节点的一个孩子所对应的子区域相交。因此，Q(n)必然满足以下递推关系：

$$Q(n) = \begin{cases} O(1) & \text{若 } n = 1 \\ 2 + 2Q(\frac{n}{4}) & \text{若 } n > 1 \end{cases}$$

这一递推关系的解为 $Q(n) = O(\sqrt{n})$。换而言之，任何垂线与一棵 kd-树中各子区域相交的数目为 $O(\sqrt{n})$。类似地也可证明：与任何水平直线相交的子区域数目也是 $O(\sqrt{n})$。而任意矩形待查询区域的边界与子区域相交的数目，上界也是 $O(\sqrt{n})$。 ❑

以上对查询时间的分析，多少显得有些悲观：在这里，为了估计与待查询矩形的任何一条边相交的子区域数目，我们估计了与矩形各边所在的（四条）直线相交的子区域数目，并以此做为前者的上界。在很多应用环境中，待查询的区域都是很小的。于是，矩形的边界很短，从而只会与少量的子区域相交。例如，要是待查询区域为[x : x] × [y : y]——此时，实际上就是询问点(x, y)是否属于点集 P——则查找的时间不会超过 $O(\log n)$。

---

① 原文此处只有两重右括号，应属排版疏漏。——译者

kd-树的性能，可以归纳为如下定理。

〖定理 5.5〗

给定由平面上任意 n 个点构成的集合 P，其对应的 kd-树将占用 $O(n)$空间，并且可以在 $O(n\log n)$时间内构造出来。使用这棵 kd-树，每次矩形区域查找所需的时间将不会超过 $O(\sqrt{n}+k)$，其中 k 为实际被报告出来的点数。

在三维或者更高维的空间中，同样可以使用kd-树。此时的构造算法与平面的情况非常类似：在根节点处，通过一张与x_1-坐标轴垂直的超平面（hyperplane），将点集划分为大致均等的两个子集。换而言之，在根节点处，整个点集是沿着第一维坐标进行划分的。在根节点的（两个）孩子处，继续沿着第二维坐标进行划分；在深度为 2 的那层，沿着第三维坐标划分；如此下去，直到深度为d-1[①]的那一层，沿着最后一维坐标划分。到达深度为d的那层之后，我们将从第一维开始重复上述过程。直到（子树中）只剩一个点时，递归才结束，而这个点将存储在一匹叶子中。与d维空间中的n个点相对应的kd-树，毕竟也是一棵拥有n匹叶子的二叉树，故只需占用$O(n)$空间。其构造过程所需的时间为$O(n\log n)$。（照例，假设d为常数。）

与平面的情况一样，d 维 kd-树中的每个节点也分别对应于某个子区域。被查找算法访问到的节点，其对应的子区域都与待查询区域真相交（properly intersecting）；被算法遍历（以报告出其中各匹叶子所对应的点）的子树，其根节点对应的子区域都完全包含于查找区域之中。可以证明，此时查询时间的上界为 $O(n^{1-1/d}+k)$。

5.3 区域树

前一节所介绍的 kd-树，每次查找需要 $O(\sqrt{n}+k)$时间。因此，若实际报告出来的点数很少，查询所消耗的时间就相对过多。本节将介绍用于矩形区域查找的另一种数据结构——区域树（range tree）。这种结构的查询时间性能更好，具体而言，是 $O(\log^2 n+k)$。为实现这一改进，需要在其它方面付出代价——存储空间将由 kd-树的 $O(n)$，上升到区域树的 $O(n\log n)$。

正如此前我们所注意到的，每次二维区域查找，实质上都是由（前后）两次一维子查找构成的，x-坐标、y-坐标方向上各一次。这就启发我们，轮流沿着 x-坐标和 y-坐标的方向，对点集进行划分——这样就得到了 kd-树。为了导出区域树，需要按照另一种方式来利用上述观察结果。

---

① 按照习惯，此处的 d 为空间的维数。——译者

设P为由平面上n个点构成的一个集合，我们准备对它进行预处理，以支持此后的矩形区域查找。设待查询区域为[x : x'] × [y : y']。首先着手找出x-坐标位于待查询矩形对应的x-区间[x : x']之内的所有点；至于y-坐标，将在以后再解决。如果只关心x-坐标，那么该查找只是一次一维区域查找。在第 5.1 节中，我们已经知道了可以如何解答这类查找——借助一棵根据所有点的x-坐标建立起来的二分查找树。对应的查找算法大致如下：在这棵树中查找x和x'，直到我们到达某个节点v_{split}，两条查找路径在这里分道扬镳。

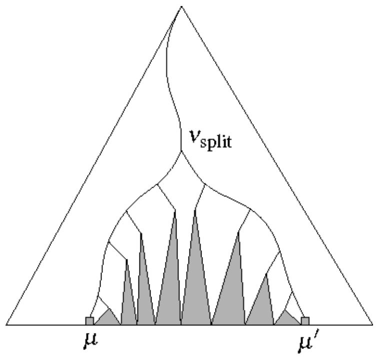


图5-12 确定v_{split}后，从这里分两路继续查找

如 图 5-12 所示，从v_{split}的左孩子开始，继续查找x。这一查找每次在某个节点v处向左前进一步，就将存储于v右子树中的点悉数报告出来。类似地，也从v_{split}的右孩子开始继续查找x'，每次在某个节点v处向右前进一步，就将存储于v左子树中的点悉数报告出来。最后，还要检查两条查找路径的终点μ和μ'，以确定它们各自对应的点是否在待查询区域之内。就其效果而言，我们只是选取出来$O(\log n)$棵子树，而且它们所存储的点合起来，恰好就是那些x-坐标落在待查询矩形所对应的x-区间之内的那些点。

在以v为根的子树中，各匹叶子所对应的点构成的（P的）一个子集，称作与v对应的**正则子集**。例如，对应于整棵树的树根的正则子集就是集合P本身。而对应于任一叶子的正则子集，就是存储于该叶子处的那个点 [①]。节点v对应的正则子集记作P(v)。刚才已经看到：x-坐标位于某个待查询区域之内的那些点所构成的子集，可以表示为$O(\log n)$个互不相交的正则子集之并——每一这类正则子集P(v)所对应的节点v，都是由算法选取出来的某棵子树的根节点。对于这类正则子集P(v)，我们并不见得会对其中所有的点都感兴趣。我们所希望的，只是将其中y-坐标落在[y : y']之内的所有点报告出来——这又将是一次一维的查找。只要对于P(v)中的所有点，我们都有一棵按照其y-坐标组织的二分查找树，那么这个问题就可以迎刃而解。这样就得到了另一个数据结构，它同样可以用于对由平面上任意n个点构成的集合P进行矩形区域查询。该结构的描述如下：

- 主树（main tree）是一棵平衡二分查找树$\mathcal{T}$，按照 P 中各点的 x-坐标，将它们组织起来。

[①] 更严格地说，应该是“与该叶子相对应的那个点所构成的单元素集合（singleton）”。——译者

- 对于 $\mathcal{T}$ 中的每个内部节点或者叶子 v，再将正则子集 P(v)中的各点按照 y-坐标，存储为一棵平衡二分查找树 $\mathcal{T}_{assoc}(v)$。节点 v 拥有一个指针，指向 $\mathcal{T}_{assoc}(v)$的根。我们称 $\mathcal{T}_{assoc}(v)$为 v 的**联合结构**（associated structure）。

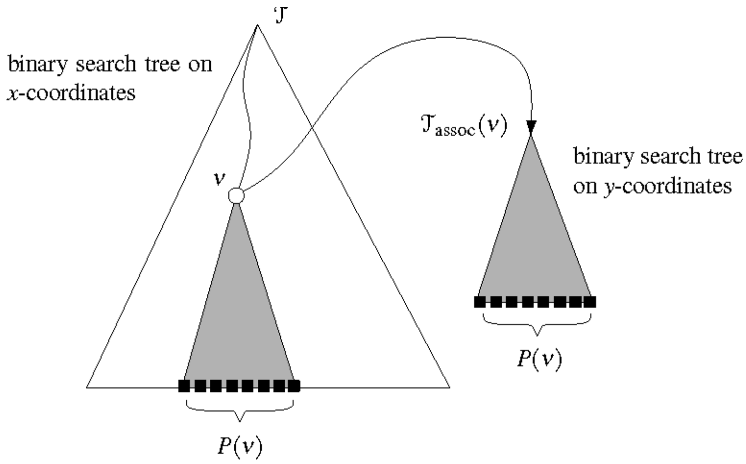


图5-13 二维区域树

上述结构被称为区域树（range tree）。图 5-13 所示的，就是一棵区域树的结构。如果某一数据结构中配有指向联合结构的指针，往往被称为多层次数据结构（multi-level data structure）。这时，主树$\mathcal{T}$被称为**第一层的树**（first-level tree），而每一联合结构都被称为**第二层的树**（second-level tree）。在计算几何中，多层次数据结构扮演了重要的角色；第 10 和 16 章将介绍更多这种例子。

可以通过下述递归算法，构造出一棵区域树。该算法的输入为集合 P := {p_1, …, p_n}，其中所有点已经按照 x-坐标排序；算法最后返回与 P 对应的一棵二维区域树的根节点。与前一节类似，这里也假设没有任何两个点有相同的 x-或者 y-坐标。在第 5.5 节，我们将去掉这一限制。

算法 BUILD2DRANGETREE (P)

Input: 平面点集 P

Output: 一棵二维区域树的树根

1. 构造联合结构：
   令 P_y 为由 P 中各点的 y-坐标组成的集合，构造一棵存储 P_y 的二分查找树 $\mathcal{T}_{assoc}$
   $\mathcal{T}_{assoc}$ 的各匹叶子，不仅要存储 P_y 中的 y-坐标，还要存储该坐标所对应的点
2. **if** (P 只包含一个点)
3. **then** 生成一匹叶子来存储这个点，并将 $\mathcal{T}_{assoc}$ 当作与 v 对应的联合结构
4. **else** (* 设 x_{mid} 为 P 中各点 x-坐标的中值 *)
   将 P 划分为两个子集 P_{left} 和 P_{right}
   P_{left} 由所有 x-坐标小于或者等于 x_{mid} 的点组成
   P_{right} 由所有 x-坐标大于 x_{mid} 的点组成
5. v_{left} ← BUILD2DRANGETREE (P_{left})

6.	v_{right} ← BUILD2DRANGETREE (P_{right})
7.	生成一个节点 v 以存储 x_{mid} 将 v_{left} 做为 v 的左孩子 将 v_{right} 做为 v 的右孩子 将 $\mathcal{T}_{assoc}$ 做为 v 的联合结构
8.	**return** v

请注意：在联合结构的叶子内，不仅要存储其对应点的 y-坐标，而且要存储这些点本身。这一点很重要，因为在对联合结构进行搜索时，我们需要的不仅是它们的 y-坐标，而且（可能）需要把这些点报告出来。

〖引理 5.6〗

给定由平面上任意 n 个点构成的点集，其对应的区域树占用 O(nlog**n)的存储空间。**

〖证明〗

对于 P 中的任一点 p，如果 p 被存放在某一联合结构之中，那么这个联合结构在 $\mathcal{T}$ 中所对应的节点，必然位于在 $\mathcal{T}$ 中由根节点通往 p 所对应的叶子的那条路径之上。也就是说，在 $\mathcal{T}$ 的同一深度层次上，点 p 只会被存储于恰好一个联合结构之中。

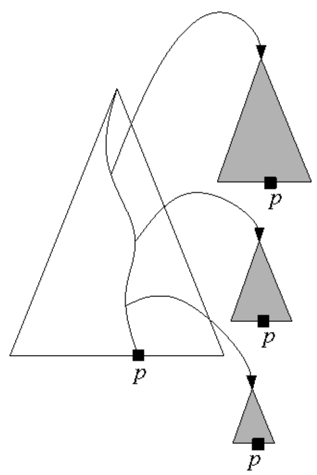


图5-14 在每一层，点p只会被存储一次

既然任何一棵一维区域树只占用线性规模的存储空间，故在 $\mathcal{T}$ 的每一深度层次上，所有节点对应的联合结构总共只占用 O(n)的存储空间。鉴于 $\mathcal{T}$ 的深度为 O(**log**n)，故其所需的总存储空间不会超过 O(n**log**n)。 ❑

根据以上的描述，算法 BUILD2DRANGETREE 的运行时间并没有达到最优的 O(n**log**n)。为了证明能够达到这一最优效率，在具体实现时必须有所讲究。如果根据 n 个未经排序的关键码来构造一棵

二分查找树，的确需要消耗 $O(n\log n)$时间。这意味着在第 1 行构造联合结构时，就需要消耗 $O(n\log n)$时间。然而，由于 P_y 中的各点已经预先经过了排序，所以可以完成得更快——只要按照自底向上的次序来构造二分查找树，总体的构造时间就将是线性的。因此，在构造的过程中，我们将维护对应于整个点集的两个列表：一个按照 x-坐标排序，另一个按照 y-坐标排序。按照这一方法，对于主树 $\mathcal{T}$ 中的每一个节点，消耗的时间将线性正比于其对应正则子集的规模。这就意味着：总体的时间将与占用空间的规模相同，即 $O(n\log n)$。鉴于预排序所需的时间也不过是 $O(n\log n)$，故整个构造算法所需的时间还是 $O(n\log n)$。

查找算法首先遴选出 $O(n\log n)$个正则子集，而这些正则子集的并，就给出了 x-坐标落在区域[x : x']之内的所有点。采用一维查找算法，也可以完成这个任务。对于每一这类子集，我们进而将其中 y-坐标落在[y : y']之内的所有点报告出来。我们可以再次通过一维查找算法来完成这项任务；不同的是，这一次要将此算法分别应用于存储各当选正则子集的联合结构。因此就其本质而言，这一查找算法与一维查找算法 1DRANGEQUERY 是相同的；唯一的差别就是，所有对 REPORTSUBTREE 的调用，都被代以对 1DRANGEQUERY 的再次调用。

```
算法 2DRANGEQUERY (T, [x : x'] × [y : y'])
输入：二维区域树 T，以及待查询区域[x : x'] × [y : y']
输出：T 中位于[x : x'] × [y : y']之内的所有点
1.   v_split ← FINDSPLITNODE (T, x, x')
2.   if (v_split 是叶子)
3.      then 检查 v_split 处所存储的点是否应该被报告出来
4.      else (* 沿着通往 x 的路径，对路径右侧的每棵子树调用一次 1DRANGEQUERY *)
5.           v ← lc(v_split)
6.           while (v 还不是叶子)
7.              do if (x ≤ x_v)
8.                    then 1DRANGEQUERY (T_assoc(rc(v)), [y : y'])
9.                         v ← lc(v)
10.                   else  v ← rc(v)
11.          检查 v 处所存储的点是否应该被报告出来
12.          (* 类似地 *)
             沿着从 rc(v_split)通往 x'的路径，对路径左侧的每棵子树所对应的联合结构
             针对区域[y : y']调用一次 1DRANGEQUERY;
             检查路径终点处的那匹叶子，看看是否应该将存储于其中的点报告出来
```

〖引理 5.7〗

若采用区域树来存储（平面上的）任意 n 个点，则与坐标方向平行的每一次矩形区域查询，只需要 $O(\log^2 n + k)$时间。这里的 k 为实际被报告出来的点数。

〖证明〗

在主树T的每个节点v处，只需要常数的时间，就可以确定查找路径应该朝何方继续前行，此时，我们也可能会调用 1DRANGEQUERY。根据〖定理 5.2〗，每一次这样的递归调用都需要$O(\log n + k_v)$时间，其中k_v为该次调用所报告出来的点数。于是，总共花费的时间就是

$$\sum_v O(\log n + k_v)$$

这一求和的范围，覆盖主树 T 中所有被访问到的节点。我们注意到：$\sum_v k_v$正好就是 k，即被报告出来的点的总数。此外，x 和 x'在主树 T 中的查找路径，长度都为 $O(\log n)$。因此，$\sum_v O(\log n) = O(\log^2 n)$。本引理得证。 ❑

二维区域树的性能，可以总结为如下定理：

〖定理 5.8〗

给定由平面上任意 n 个点构成的集合 P。对应于 P 的一棵区域树占用 $O(n\log n)$的存储空间，并且可以在 $O(n\log n)$时间内构造出来。对这棵区域树进行查找，可以在 $O(\log^2 n + k)$时间内，从 P 中报告出落在任一矩形待查询区域之内的所有点，其中 k 为实际被报告出来的点数。

〖定理 5.8〗所声称的查询时间，还可以进一步改进至$O(\log n + k)$，为此需要借助于一种称作**分散层叠**（fractional cascading）的技术。第 5.6 节将介绍这一技术。

5.4 高维区域树

二维区域树可以直接推广为更高维的区域树。在这里只讨论其算法的主体框架。

考虑d维空间中的任一点集P。我们按照其中各点的第一维坐标，构造出一棵平衡二分查找树。第一层次的这棵树也就是主树，对于其中的任一节点v，在以v为根节点的子树中，所有叶子各自对应的点，合起来构成了与v对应的正则子集P(v)。对每个节点v，我们还为其构造一个联合结构$T_{assoc}(v)$；第二层次的每棵树$T_{assoc}(v)$，都是对应于P(v)中某个点的一棵(d-1)维区域树——此时其中各点的坐标都

限制在后(d-1)维上 [①]。递归地运用前面的方法，就可以构造出这棵(d-1)维区域树：它将是按照各点第二维坐标组织的一棵平衡二分查找树，其中各个节点都通过一个指针，指向一棵(d-2)维区域树，这棵区域树存储了该节点的子树 [②]中的所有点——同样地，这些点都被限制在后(d-2)维上。当各点都被限制于其最后一维上时，这个递归的过程才结束；此时的各点，都被存储于某棵一维区域树——即平衡二分查找树——中。

相应的查找算法与二维情形也很类似。

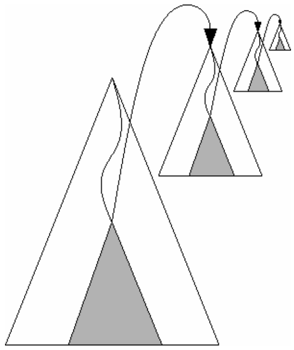


图5-15 在高维区域树中逐层查找

在第一层（主）树中，我们找出$O(\log n)$个节点——它们各自对应的正则子集合并起来，就是第一维坐标落在待查询区域之内的所有点。如 图 5-15 所示，对这些正则子集，还要分别做进一步查找，也就是在其对应的第二层结构中进行查找。在每个第二层结构中，我们再找出$O(\log n)$个正则子集。这样，在第二层结构中被找出的正则子集总数为$O(\log^2 n)$。这些正则子集的并，就是第一、二维坐标（同时）位于待查询区域之内的所有点。接下来，要针对第三维坐标，对存储这些正则子集的第三层结构进行查找，……，如此下去，直到一维的树结构。在这一层上，我们要筛选出最后一维坐标落在待查询区域之内的所有点，并且将它们报告出来。根据这种方法，可以得出如下结论：

〖定理 5.9〗

给定由 d 维空间中任意 n 个点构成的集合 P，$d \geq 2$。对应于 P 的一棵区域树占用 $O(n\log^{d-1} n)$的存储空间，并且可以在 $O(n\log^{d-1} n)$时间内构造出来。对这棵区域树进行查找，可以在 $O(\log^d n + k)$时间内，从 P 中报告出落在给定（超）矩形待查询区域之内的所有点，其中 k 为实际被报告出来的点数。

〖证明〗

一棵对应于d维空间中n个点的区域树，其构造时间记作$T_d(n)$。根据〖定理 5.8〗，有$T_2(n) = O(n\log n)$。在构造一棵d维区域树的过程中，我们需要构造一棵平衡二分查找树，这需要

① 也就是说，每个点都被替换为它在此(d-1)维子空间中的投影。——译者

② 即以该节点为根的那棵子树。——译者

$O(n\mathbf{log}n)$时间；我们还要构造所有对应的联合结构。在第一层树中，在同一深度的所有节点处，（P中的）每个点都被存储而且仅被存储于一个联合结构之中。对处于同一深度的所有节点而言，分别构造它们所对应的联合结构，总共需要$O(T_{d-1}(n))$时间，这也就是为了构造对应于根节点的联合结构所需要的时间——这是因为，构造时间不会低于线性量级。因此，整体的构造时间必然满足：

$$T_d(n) \quad = \quad O(n\mathbf{log}n) + O(\mathbf{log}n)\times T_{d-1}(n)$$

鉴于 $T_2(n) = O(n\mathbf{log}n)$，上述递推关系的解应为 $O(n\mathbf{log}^{d-1}n)$。按照相同的分析方法，也可以得出所需存储空间的上界。

对应于 d 维空间中任意 n 个点的一棵区域树，其查询时间记作 $Q_d(n)$——这里不计为报告各点所需的时间。每次对 d 维区域树的查找，首先需要对第一层的树进行查找，这一步所需的时间为 $O(\mathbf{log}n)$；接下来，还需要对对数量级棵(d-1)维区域树进行查找。这样，就有：

$$Q_d(n) \quad = \quad O(\mathbf{log}n) + O(\mathbf{log}n)\times Q_{d-1}(n)$$

在这里，$Q_2(n) = O(\mathbf{log}^2n)$。于是，很容易就可以得出上述递推关系的解为 $Q_d(n) = O(\mathbf{log}^dn)$。

最后，还要计入为报告各点所需花费的时间，这部分时间不会超过 $O(k)$。这样，就得出了每次查询所需的时间。 ❑

与二维的情况一样，这里的查询时间也可以有一个对数因子的改进——参见第 5.6 节。

5.5 一般性点集

到现在为止，我们一直强加了一个限制：任何两点的 x-和 y-坐标不得相同——这一限制极其不切实际。幸运的是，这一点可以很容易得到补救。最关键的就是要注意到：我们从来都没有假定坐标的数值必须是实数。我只要求它们来自某个全序域（totally ordered universe）。只要满足这一条件，就可以对任意两个坐标进行比较，并可以计算中值。因此，我们将运用如下技巧。

我们原先表示为实数的坐标，被替换为来自所谓合成数空间（composite number space）的元素。这种空间中的每个元素都由一对实数组成。对应于 a 和 b 两个实数的合成数，记作(a|b)。按照字典序（lexicographical order），我们可以在合成数空间中定义一个全序（total order）。这样，对于任何两个合成数(a|b)和(a'|b')，都有

$$(a|b) < (a'|b') \quad \Leftrightarrow \quad a < a' \text{ 或者 } (a = a' \text{ 且 } b < b')$$

现在，考虑由平面上任意 n 个点构成的集合 P。尽管该集合中的点必然互异，但是可能会有很多个点的 x-或 y-坐标相同。我们将每个点 $p := (p_x, p_y)$，替换为一个以合成数为坐标的点 $\hat{p} := ((p_x|p_y), (p_y|p_x))$。如此就得到了由 n 个点构成的一个新集合 $\hat{P}$。在 $\hat{P}$中，任意两个点的第一个坐标必然不同；第二个坐标亦是如此。按照以上定义的序，我们现在就可以为 $\hat{P}$构造一棵 kd-树或者二维的区域树。

现在假设我们希望报告出 P 中位于 $R := [x : x'] \times [y : y']$之内的所有点。为此，必须对刚才为 $\hat{P}$构造的树进行查找。也就是说，必须同时将待查询区域也变换到新的合成数空间。这一变换定义如下：

$$\hat{R} \;:=\; [(x|-\infty) : (x'|+\infty)] \times [(y|-\infty) : (y'|+\infty)]$$

接下来，只需证明此方法的正确性。也就是说，需要证明：在针对 $\hat{R}$进行查找时，从 $\hat{P}$中报告出来的点，不多不少恰好对应于 P 中落在 R 之内的所有点。

〖引理 5.10〗

对于任意的一个点 p 以及一个矩形区域 R，都有：

$$p \in R \iff \hat{p} \in \hat{R}$$

〖证明〗

设 $R := [x : x'] \times [y : y']$，$p := (p_x, p_y)$。根据定义，p 落在 R 之内当且仅当“$x \leq p_x \leq x'$ 且 $y \leq p_y \leq y'$”。易见，此充要条件被满足当且仅当“$(x|-\infty) \leq (p_x|p_y) \leq (x'|+\infty)$且$(y|-\infty) \leq (p_y|p_x) \leq (y'|+\infty)$”，也就是当且仅当 $\hat{p}$位于 $\hat{R}$之内。 ❑

由此可知，我们的方法的确是正确的——通过这种方法，可以得到正确的查询结果。请注意：实际上并不需要将变换后的点存储下来——只要能够在合成数空间中对任意两个 x-坐标或者 y-坐标进行比较，就可以只存储原始的那些点。

在更高维的空间中，同样可以采用合成数的方法。

5.6 *分散层叠

第 5.3 节针对平面矩形区域查找的问题，描述过一种被称为区域树的数据结构，其查询时间为 $O(\log^2 n + k)$。（这里，n 为该数据结构所存储的点的总数，而 k 则是最终被报告出来的点的数目。）本节将要介绍一种被称为分散层叠（fractional cascading）的技术，可使查询时间降至 $O(\log n + k)$。

我们首先回顾一下区域树的工作过程。对应于平面点集 P 的一棵区域树，是一个分为两层的结构。其中的主树是一棵关于各点 x-坐标的二分查找树。对应于主树中的每个节点 v，都有一个联合

结构 $\mathcal{T}_{assoc}(v)$，它是一棵关于 P(v)中各点 y-坐标的二分查找树（此处的 P(v)表示 v 对应的正则子集）。针对任一矩形区域[x : x'] × [y : y']的查询，进行过程如下：首先，在主树中确定一组共 $O(\log n)$个节点，它们所对应正则子集的并，给出了 x-坐标落在区域[x : x']之内的所有点。接下来，针对区域[y : y']，分别对这些节点所对应的联合结构进行查找。对任何联合结构 $\mathcal{T}_{assoc}(v)$的查找，就是一次一维区域查找，故只需 $O(\log n + k_v)$时间，其中 k_v 是报告出来的点数。因此，总的查询时间为 $O(\log^2 n + k)$。

如果能够在 $O(1 + k_v)$时间内完成对单个联合结构的查找，那么总体查询时间就可以降低到 $O(\log n + k)$。然而，如何才能做到这样呢？一般而言，要想在 $O(1 + k)$时间（这里的 k 同样是被报告出来的点数）内完成一次一维区域查找是不可能的。我们能够加以利用的一个条件是，这里需要对**同一区域**进行**多次**一维查找——因此，可能借助前一次查找的结果来加速后续的查找。

首先通过一个简单的例子，来说明分散层叠的构思。设 S_1 和 S_2 分别为由对象组成的两个集合，其中每个对象都有一个实数类型的关键码。这两个集合经过排序，分别被存储于两个数组 A_1 和 A_2 之中。现在假设，要求将 S_1 和 S_2 中关键码位于某个待查询区间（query interval）[y : y']之内的所有对象都报告出来。我们可以这样来做：在 A_1 中对y进行二分查找，找出其中大于或等于y的最小关键码。从这个位置开始，沿着数组 A_1 向右前行，直到遇见第一个大于y'的关键码——在此期间途经的每个对象都要报告出来。类似地，S_2 中符合要求的对象也可以被报告出来。如果被报告出来的对象总数为k，那么查询时间就是 $O(k)$，再加上（在 A_1 和 A_2 中分别进行的）两次二分查找所需的时间。然而，如果在比较 S_1 和 S_2 对应的关键码集合之后，发现后者是使前者的一个子集，那么我们就可以通过下面的方法，避免第二次二分查找。我们给数组 A_1 的每个元素，都配备一个指针，指向 A_2 的某个元素：如果 $A_1[i]$所存储对象的关键码为 y_i，那么我们为它配备的指针，将指向 A_2 中不小于 y_i 的最小关键码。若 A_2 中没有这样一个关键码，则将 $A_1[i]$对应的这个指针置为nil。图 5-16 对此做了形象的解释。（此图中只标出了关键码，而不是对应的对象。）

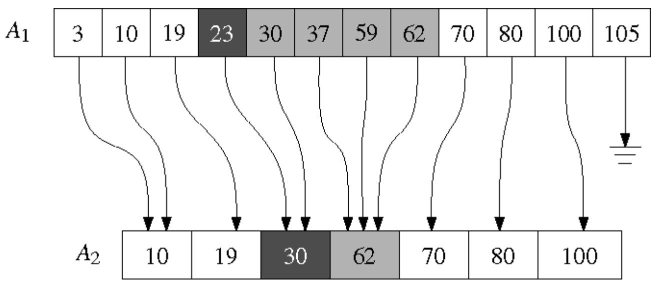


图5-16 增加指针以加速查找

现在，如何才能报告出 S_1 和 S_2 中落在同一待查询区间[y : y']之内的所有对象呢？报告 S_1 中的有关对象，方法没有变化：在 A_1 中对 y 进行二分查找，然后沿着 A_1 朝右方前行，直到第一个大于y'的关键码。为了报告 S_2 中的有关对象，可以采用如下方法。设在 A_1 中对 y 的查找终止于 A[i]。亦

即，A[i]的关键码是 S_1 中不小于 y 的最小关键码。既然 S_2 的关键码集合是 S_1 的关键码集合的子集，故 A[i]的指针必定指向 S_2 中不小于 y 的最小关键码。因此，就可以沿着这个指针（找到 A_2 中对应的元素），然后沿着 A_2 朝右方前行。这样，原来对 A_2 进行的二分查找就可以省去，而为了报告出 S_2 中的有关对象，只需要 $O(1+k)$时间（其中 k 为实际被报告出来的对象数目）。

图 5-16 给出了这种查找的一个实例。我们对区域[20 : 65]进行查找。首先，在A_1 中通过二分查找，确定（不小于 20 的最小关键码）23 的位置。从此处出发，朝右方前进，直到发现第一个大于 65 的关键码。在此期间所经过的对象，其关键码都在指定区域之内，因此这些对象将被报告出来。然后，沿着 23 所对应的指针转入A_2。这样，就到达了关键码为 30 的元素，在A_2 中，30 是不小于 20 的最小关键码。从这个位置出发，同样朝右方前进，直到发现第一个大于 65 的关键码，在此过程中，所有关键码位于指定区域之内的对象都会被报告出来。

现在回到区域树。这里最重要的一个观察结果就是：正则子集 P(lc(v))和 P(rc(v))都是 P(v)的子集。因此，我们可以采用上面所构想的方法，来加速查找过程。其实现细节略显复杂——因为，现在需要处理的是 P(v)的两个子集，而我们需要同时对二者，而不是对其中之一进行快速访问。平面上任意 n 个点构成集合 P，设 $\mathcal{T}$ 为与 P 对应的区域树。每一正则子集 P(v)都被存储于某个联合结构之中。然而，我们不再像第 5.3 节那样将联合结构组织成一棵二分查找树，而是组织成一个数组 A(v)。每一数组都按照 y-坐标排序。此外，A(v)中的每一元素都配有两个指针，分别指向 A(lc(v))和 A(rc(v))。更准确地说，这两个指针的设置方法如下。假设在 A(v)[i]处存储的是点 p。于是，我们为 A(v)[i]设置一个指针，指向 A(lc(v))中的一个元素，该元素所存储的点 p'的 y-坐标，（在 A(lc(v))中）是不小于 p_y 的最小者。正如上面所指出的，P(lc(v))是 P(v)的一个子集。因此，若在 P(v)所存储的诸点中，p 的 y-坐标是不小于某一 y 值的最小者，则在 P(lc(v))所存储的诸点中，p'的 y-坐标也是不小于 y 的最小者。指向 A(rc(v))的指针的设置方法相同：在 A(rc(v))所存储的诸点中，该指针所指（位置处所存储）的那个点，其 y-坐标是不小于 p_y 的最小者。

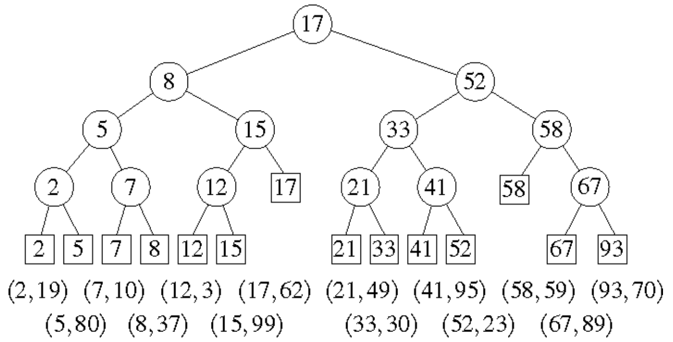


图5-17 层次化区域树的主树：这里仅仅画出了各叶子的x-坐标，各叶子处所存储的点则被标注于下方

做过上述改动之后的区域树，称作层次化区域树；在 图 5-17 和 图 5-18 中显示的，就是这样的一个例子。（其中，各数组所画的位置，分别对应于其在（主）树中所关联节点的位置：最顶层的那个数组对应于根节点；其下面的左侧的那个数组对应于根节点的左孩子；依此类推。图中并没有画出所有的指针。）

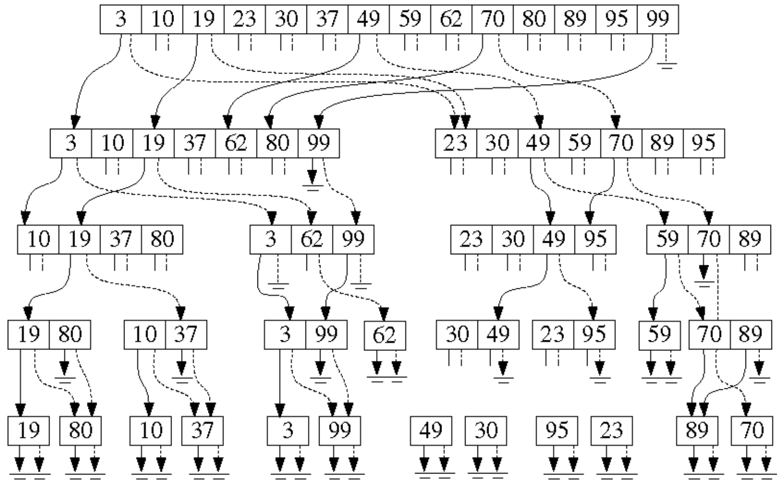


图5-18 与主树中各节点相关联的数组：各数组所对应正则子集中的诸点，都按照y-坐标排序（这里并没有画出所有的指针）

让我们来看看，在一棵层次化区域树中，如何针对某一区域$[x : x'] \times [y : y']$进行查找。与原来一样，我们在主树T中分别查找x和x'，从而找出$O(\log n)$个节点——它们对应的正则子集合并起来，正好给出了x-坐标落在$[x : x']$之内的所有点。这些节点是这样找出来的。设两条查找路径在节点v_{split}处分道扬镳。我们所希望找出的那些节点，都是v_{split}的某些后代：它们或者是通往x的查找路径向左转向处的右孩子，或者是通往x'的查找路径向右转向处的左孩子[①]。在v_{split}处，我们可以找到$A(v_{split})$的入口位置，该元素的y-坐标是其中不小于y的最小者。通过二分查找，这可以在$O(\log n)$时间内完成。至于各关联数组中具有不小于y的最小y-坐标的元素，在主树内继续对x和x'进行查找的过程中，可以跟踪记录下这些元素的位置。借助于如下在各数组中设置的指针，可以在常数时间内对这些元素的位置进行维护与更新。设v就是我们所筛选出来的$O(\log n)$个节点之一。我们必须从存储于$A(v)$处的各点中，找出y-坐标落在$[y : y']$之内的所有点，并报告出来。为此，只需首先找到其中不小于y的最小y-坐标；然后，从这一位置出发（向右）前行，逐个报告途经的每一元素，直到遇见第一个大于y'的y-坐标。第一个点可以在常数时间内找到——因为，parent(v)位于查找路径之上，而且对于查找路径沿途各节点所对应的数组中具有不小于y的最小y-坐标的点，我们已经做了跟踪记录。因此，为了将$A(v)$中y-坐标落在$[y : y']$之内的所有点报告出来，只需$O(1 + k_v)$时间，其中k_v为在节点v处实际被报告的点数。这样，总的查询时间就变成了$O(\log n + k)$。

[①] 此处为原文直译，不易理解。按照中文习惯，这些节点应该定义为“以沿 v_{split} 通往 x 或 x'的查找路径之上某个节点为父亲，本身却不属于这两条路径的节点”。——译者

对于更高维的区域树，采用分散层叠技术同样可以将其查询时间降低一个对数因子。也许你还记得：在求解 d 维区域查找时，首先要筛选出 $O(\mathbf{log}n)$个正则子集——它们包含了第 d 维坐标落在待查询区域之内的所有点；接下来，再分别对这些正则子集进行(d-1)维的区域查找。这些(d-1)维的区域查找将采用相同的方法，递归地进行。直到二维区域查找阶段，递归才终止；而二维的区域查找，则可以通过以上方法加以解决。由此可以得出如下定理：

〖定理 5.11〗

在 d(≥2)维空间中，给定由任意 n 个点构成的集合 P。总可以在 $O(n\mathbf{log}^{d-1}n)$时间内，构造出一棵与 P 对应的层次化区域树，其占用的存储空间为 $O(n\mathbf{log}^{d-1}n)$。借助于这棵区域树，可以在 $O(\mathbf{log}^{d-1}n + k)$时间内，报告出 P 中落在某矩形待查询区域之内的所有点，其中 k 为实际被报告出来的点数。

5.7 注释及评论

二十世纪七十年代，计算几何呱呱落地。即便在那时，正交区域查找就已经是该领域中最重要的问题之一，并有许多人致力于研究它。他们的工作成果颇为丰富，下面仅介绍其中一二。

在解决正交区域查找问题方面，人们采用的第一种数据结构是四叉树（quadtree）——在后面的第 14 章讨论网格划分问题时，将对其做一介绍。不幸的是，四叉树在最坏情况下的性能相当差。Bentley[44]于 1975 年提出的kd-树，是对四叉树的改进。Samet在其专著 [333][334]中，对四叉树、kd-树及其应用做过详细的讨论。数年之后，有多人 [46][251][261][387]分别独立地提出了区域树。而借助于分散层叠技术将查询时间改进至$O(\mathbf{log}n + k)$的方法，则是由Lueker[261]和Willard[386]提出的。分散层叠不仅可以应用于区域树，实际上，在很多需要根据同一关键码进行多次查找的场合，都同样适用。Chazelle和Guibas[105][106]对这一技术做了透彻的介绍。分散层叠还可以应用于动态设置 [275]。Chazelle[87]提出了一种修改后的层次化区域树，这是解决二维区域查找问题最有效的数据结构；他成功地将存储空间降至$O(n\mathbf{log}n/\mathbf{loglog}n)$，同时保持了$O(\mathbf{log}n + k)$的查询时间。Chazelle[90][91]还证明了这一结果已经是最优的。如果待查询区域在某一侧是无界的（例如，其形式为$[x : x'] \times [y : +\infty]$），那么借助于一棵优先查找树（priority search tree，参见第 10 章），只需线性的空间，就能够实现$O(\mathbf{log}n)$的查询时间。对于更高维的情况，正交区域查找的最佳结果同样是由Chazelle[90]得出的：他提出一种结构，使用$O(n(\mathbf{log}n/\mathbf{loglog}n)^{d-1})$的存储空间，就可以在对数多项式的查询时间（polylogarithmic query time）①内回答一次d维查询。同样地，这一结果也是最优的。另外，可以用存储空间换取查询时间，也可以用查询时间来换取存储空间 [338][391]。

有关下界（lower bound）的结论，只对某些特定的计算模型是有效的。因此，在某些特殊情况

① 即 $O(\log^c n)$，其中 c > 1 为常数。——译者

下，还可以进一步提高效率。比如，Overmars[300]提出了一种解决区域查找问题的更加有效的数据结构：如果所有点的位置都限制在$U \times U$的网格之上，那么查询时间将是$O(\log\log U + k)$或者$O(\sqrt{U} + k)$——具体是那一结果，取决于允许的预处理时间。这些结果借用了早年由Willard[389][390]提出的一种数据结构。只要对象的坐标被限制在网格点[①]上，那么相对于一般情况而言，计算几何中的许多问题都有更好的时间复杂度上界[②]。这方面的例子包括最近邻查找 [224][225]、点定位 [287]以及线段求交 [226]等问题。

在数据库领域，区域查找被认为是三种基本类型的多维查找中最为常见的一类。其余的两类分别是**完全匹配查询**（exact match query）与**部分匹配查询**（partial match query）。所谓完全匹配查询，就是这样的一类查找：其属性值（坐标）符合某种特定条件的那些对象（点），是否出现在数据库[③]中？针对完全匹配查询问题，一种显而易见的数据结构就是平衡二分查找树——比如，这棵树可以按照坐标的字典序（lexicographical order）进行组织。采用这种结构，每一完全匹配查询都可以在$O(\log n)$时间内完成。然而，随着维度——即属性数目——的增加，为了更好地评价查找的有效性，更好的方法不仅应该考虑到（对象个数）n，同时还要兼顾空间的维数d。按照这种标准，若采用二分查找树进行完全匹配查询，则查询时间应该是$O(d*\log n)$——因为，此时每比较一对点需要花费$O(d)$时间。这一复杂度很容易就可以降至$O(d + \log n)$——而这一复杂度已经是最优的了。所谓部分匹配查询，是指给定对应于某些坐标方向的数值，然后找出对应坐标为这些特定数值的所有点。比如在平面上，部分匹配查询可能只关心x-坐标，也可能只关心y-坐标。从几何角度来理解，每次部分匹配查询实质上就是找出位于一条水平（或者垂直）线上的所有点。利用一棵d维kd-树，每次针对$s(<d)$个坐标的部分匹配查询[④]，都可以在$O(n^{1-s/d} + k)$时间内完成，其中k为实际被报告出来的点数 [44]。

在很多实际应用中，输入并不是一个点集，而是一组诸如多边形之类的对象。如果这时我们希望报告出完全落在某一待查询区域$[x : x'] \times [y : y']$之内的所有对象，那么就可以将这类查找转换为对更高维空间中某一点集的查找——参见 习题 5.13。人们也经常会要求找出部分落在某个区域之内的所有对象。这个特殊的问题被称为**截窗问题**（windowing problem），第 10 章将讨论这一问题。

区域查找问题还有其它一些变种，比如允许采用其它类型（如圆形或者三角形）的待查询区域。采用所谓的划分树（partition tree），可以解决很多的此类问题。第 16 章将讨论这种结构。

---

① 即每个点的所有坐标都必须是整数。——译者

② 即最坏情况下的复杂度更低。——译者

③ 从计算几何的角度来看，数据库就相当于 d 维空间。——译者

④ 从几何角度来看，也就是在 d 维空间中查找位于某个正交(d-s)维子空间中的所有点。——译者

5.8 习题

习题 5.1 在估计 kd-树的查询时间时，我们得出了如下递推关系：

$$Q(n) = \begin{cases} O(1) & 若 n = 1 \\ 2 + 2Q(\frac{n}{4}) & 若 n > 1 \end{cases}$$

试证明，该递推式的解为 $Q(n) = O(\sqrt{n})$。另外，对任一（足够大）的 n，试构造出由 n 个点构成的一个集合以及一个矩形查找区域，以说明$\Omega(\sqrt{n})$也是对 kd-树查找的（时间复杂度）下界。

习题 5.2 试给出在 kd-树中插入和删除点的算法。在你的算法中，不必考虑如何对该结构做再平衡化处理。

习题 5.3 第 5.2 节曾经指出，也可以使用 kd-树来存储高维空间中的点集。设 P 为由 d 维空间中任意 n 个点构成的一个集合。在下面的 a、b 部分中，你都可以将 d 视作一个常数。

a. 试给出一个算法，构造对应于 P 中所有点的一棵 d 维 kd-树。试证明，该树占用线性的存储空间，而该算法的运行时间为 $O(n\log n)$。

b. 试给出执行 d 维区域查找的查找算法。试证明，其查询时间的上界为 $O(n^{1-1/d} + k)$。

c. 试证明，占用的存储量将随着 d 线性递增——也就是说，若不将 d 视为常数，则存储总量为 $O(dn)$。同样地，也请给出构造时间与查询时间随 d 变化的关系。

习题 5.4 kd-树可以用来解决部分匹配查询。对二维部分匹配查询而言，也就是给定对应于某个坐标分量的一个数值，要求找出该坐标分量等于该数值的所有点。就更高维的情况而言，我们将给定对应于若干坐标分量的一组数值。在此，我们允许多个点的某些坐标分量数值相同。

a. 试证明，借助于二维 kd-树，可以在 $O(\sqrt{n} + k)$时间内回答每次部分匹配查询，其中 k 为实际被报告出来的点数。

b. 试说明，应该如何利用二维 kd-树来解答部分匹配查询。相应的查询时间是多少？

c. 试给出一种数据结构，它占用线性的存储空间，并能够在 $O(\log n + k)$时间内解答二维部分匹配查询。

d. 试证明，借助于一棵 d 维 kd-树，我们能够在 $O(n^{1-s/d} + k)$时间内，解答一次 d 维部分匹配查询，其中 $s(<d)$为数值被指定的坐标分量的数目。

e. 试给出一种数据结构，它占用线性的存储空间，并能够在 $O(\log n + k)$时间内解答 d 维部分匹配查询。**提示**：使用某种数据结构，其占用的存储空间随着 d 按指数速度递增（更准确地说，该结构占用 $O(d \cdot 2^d \cdot n)$空间）。

习题 5.5 即使待查询区域是矩形之外的其它形状，算法SEARCHKDTREE也依然适用。例如，即使待查询区域为三角形，对应的查找也依然可以得出正确的结果。

a. 试证明，针对三角形的区域查找，最坏情况下的求解时间是线性的；即便实际上

没有任何点会被报告出来，亦是如此。**提示**：令存储于kd-树中的所有点都落在直线y = x上。

b. 假设需要某种数据结构来求解三角形区域查找，然而三角形的边只能是水平的、垂直的或者斜率为+1 或-1。试设计一个占用线性存储空间的数据结构，在$O(n^{3/4} + k)$时间内求解这类查找，其中k为实际被报告出来的点数。**提示**：在平面上选取四条轴，然后使用四维kd-树。

c. 试将查询时间改进至$O(n^{2/3} + k)$。**提示**：首先解答 习题 5.4。

习题 5.6 试分别给出在一棵区域树中插入和删除点的相应算法。你不必考虑如何对该结构做再平衡化处理。

习题 5.7 在证明〖引理 5.7〗时，我们对各联合结构所对应的查询时间做了相当粗略的估算，当时只是（笼统地）声明，这部分时间不会超过$O(\mathbf{log}n)$。而在实际中，这部分查询时间将取决于在联合结构中实际存储的点数。将正则子集P(v)中包含的点数记作n_v。这样，总共花费的时间就是：

$$\sum_{v}\Theta(\log n_v + k_v)$$

这里的求和，要覆盖主树$\mathcal{T}$中所有被访问到的节点。试证明，这样计算出来的复杂度（确界）依然是$\Theta(\mathbf{log}^2 n + k)$。（换而言之，这里更为精细的估算，只能改进复杂度界限的常数，而不会动摇其数量级。）

习题 5.8 〖定理 5.8〗指出，与由平面上任意n个点构成的点集相对应的区域树，占用的存储空间为$O(n\mathbf{log}n)$。若只存储对应于主树中某些（特定）节点的联合结构，则可以降低存储的消耗。

a. 比如，可以只给深度为 0、2、4、6、…的那些节点，分别设置一个联合结构。试说明，可以如何修改原来的查找算法，从而仍然可以正确地解答每次区域查找。

b. 试分析此数据结构的空间复杂度及其查询时间。

c. 也可能只给深度为 0、$\lfloor\frac{1}{j}\mathbf{log}n\rfloor$、$\lfloor\frac{2}{j}\mathbf{log}n\rfloor$、…的那些节点分别设置一个联合结构，其中$j \geq 2$为常数。试分析此数据结构的空间复杂度及其查询时间。要求用n和j来表示其复杂度。

习题 5.9 使用本章所介绍的数据结构，我们可以这样来确定某个特定点(a, b)是否属于某个给定的点集：将待查询区域设为$[a : a] \times [b : b]$，然后进行区域查找。

a. 试证明，对一棵 kd-树进行上述区域查找，需要 $O(\mathbf{log}n)$时间。

b. 若换成一棵区域树，其时间复杂度上界又将是多少？试证明你的结论。

习题 5.10 在某些应用中，人们可能只对落在某个区域内点的数目感兴趣，而不需要将它们逐一报告出来。此类查找通常被称为区域基数查询（range counting query）。这种情况下，或许你会希望将查询时间复杂度中的 $O(k)$部分剔除掉。

a. 试说明，应该如何对一维区域树进行修改，从而使我们能够在 $O(\mathbf{log}n)$时间内完

成一次区域基数查询。试证明这一时间复杂度上界。

b. 试利用上述一维问题的结果，说明如何才能够在 $O(\mathbf{log}^d n)$时间内完成一次 d 维区域基数查询。试证明这一时间复杂度。

c.* 试说明，如何借助分散层叠技术，将二维以及更高维区域基数查询的时间复杂度降低一个 $O(\mathbf{log} n)$因子。

习题 5.11 给定由互不相联的n条水平线段构成的一个集合S_1，以及由互不相联的m条垂直线段构成的一个集合$S_2$①。试给出一个平面扫描线算法，在$O((n+m)\mathbf{log}(n+m))$时间内统计出$S_1 \cup S_2$中线段之间交点的数目。

习题 5.12 第 5.5 节曾经证明，通过合成数（composite number）方法，我们可以处理平面上多个点的某一坐标相同的情况。试将此概念推广至 d 维空间。为此，你需要定义出对应于 d 个数字的合成数，以及不同合成数之间的次序。然后，请说明应该如何根据这一次序，对点 $p := (p_1, ..., p_d)$和区域 $R := [r_1 : r_1'] \times \cdots \times [r_d : r_d']$进行变换，以保证：$p \in R$ 当且仅当变换后的点位于变换后的区域内。

习题 5.13 在某些应用中，人们要做的区域查找可能针对的是一些对象，而不是一些点。

a. 给定由平面上与坐标轴平行的 n 个矩形所构成的一个集合 S。我们希望将 S 中完全落在某个待查询矩形$[x : x'] \times [y : y']$内的所有矩形报告出来。试给出能够用以解决此问题的一种数据结构，该结构占用 $O(n\mathbf{log}^3 n)$的存储空间，对应的查询时间为$O(\mathbf{log}^4 n + k)$，其中 k 为实际被报告出来的矩形数目。**提示**：将此问题转化为某一更高维空间中的正交区域查找问题。

b. 给定由平面上任意 n 个多边形构成的一个集合 P。同样地，试给出一种占用$O(n\mathbf{log}^3 n)$存储空间的数据结构，它可以在 $O(\mathbf{log}^4 n + k)$时间内，报告出 P 中完全落在该待查询矩形内的所有多边形，k 为实际被报告出来的多边形数目。

c.* 试将你（在 a 和 b 中）实现的查询时间，都改进到 $O(\mathbf{log}^3 n + k)$。

习题 5.14* 试证明：〖定理 5.11〗所声称的存储空间及构造时间的复杂度上界，的确为$O(n\mathbf{log}^{d-1} n)$。

① 这里，所有线段都来自同一平面。——译者

6

点定位：找到自己的位置

本书的大部分内容，都是在欧洲写成的。更准确地说，我们写作的地理位置非常接近于东经 5º6'、北纬 52º3'那一点。那是什么地方？只要手头有一份欧洲地图，你就可以自己来找到这个位置：按照地图边框上的标尺，你将会发现，上述坐标所对应的地方属于一个名叫“荷兰”的国家。

按照上述方式可以回答形式如下的点定位查询（point location query）：给定一张地图以及由坐标指定的一个查询点q，在图中找出q所处的子区域。当然，这里所说的“地图”，只不过是将整个平面分割成多个区域之后的结果，也就是第 2 章所定义的平面子区域划分（planar subdivision）。

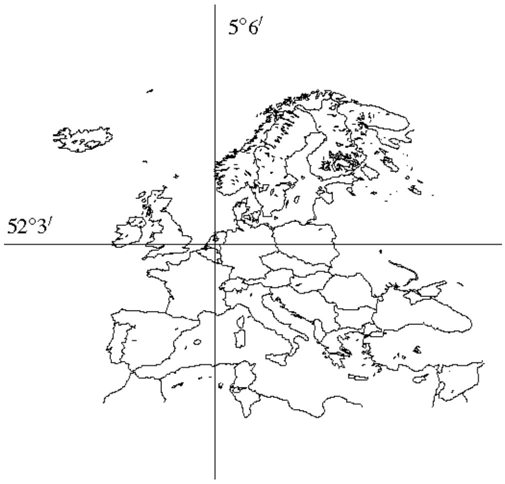


图6-1 地图上的点定位

在许多不同的场合中，都会提出这类点定位查询的问题。假设你正在海上航行，而且随处可能会遇到浅滩和险流。为保证航行的安全，你必须了解自己当前所处位置的水流状况。幸运的是，有现成的航海图标出了海上各处的水流状况。你应按照如下方法来使用航海图。首先，需要确定你当前的位置。不久以前，我们还只能依靠星辰或太阳，外加一个准确的计时器来做到这一点。然而今天，你已经可以更容易地来确定自己的位置——只要到市场上买个个头不大的盒子，你就可以借助各种卫星提供的信息，随时确定自己所处的位置。在确定了自己所处的位置之后，你需要在航海图上找到对应的点，然后才能查看那里的水流情况，或者看看自己到底处于海洋中的那块区域。

更进一步地，还可以让最后一步变成自动的——将航海图转换为电子格式，进而利用计算机来为你确定方位。这样，无论何时何地，计算机都能够动态地显示出你现在所处位置的水流情况（你所拥有的也可能是电子格式的其它专题图，这些情况下，你也可以获得相应的其它信息）。在这种情况下，我们可能会拥有一套相当详细的专题图，我们希望能够频繁地进行点定位查询，从而在航行的过程中，不断地更新显示的当前信息。这就意味着，我们希望对航海图进行预处理（preprocessing），然后将有关信息组织为某种数据结构，从而使得点定位查询可以很快完成。

点定位问题可以出现在不同的领域。假设我们想要实现一个通过屏幕显示地图的交互式地理信息系统。用户只要用鼠标点击某个国家，就可以查阅该国的信息。随着鼠标在屏幕的不同位置之间移动，该系统应该能够始终提供鼠标所指国家的名称。显然，对于屏幕上所显示的那张地图而言，这就是一个点定位问题；在这里，鼠标的位置就相当于查询点。这类查询进行的频率很高——毕竟，我们希望能够实时地更新屏幕上的信息——因此，这种查询必须很快得到回答。这样，我们就再次需要借助某种数据结构，来支持这种快速的点定位查询。

6.1 点定位及梯形图

令 S 为一个包含 n 条边的平面子区域划分（subdividsion）。所谓的平面点定位（planar point location）问题，要求将 S 存储为某种形式，以使得我们能够回答形式如下的查询：对任一查询点 q，在 S 中找出 q 所处的那张面 f。要是 q 碰巧落在某条边上，或者恰好与某个顶点重合，查询算法也必须返回这些信息。

为加深对该问题的理解，首先来看一种非常简单，却可用来进行点定位查询的数据结构。

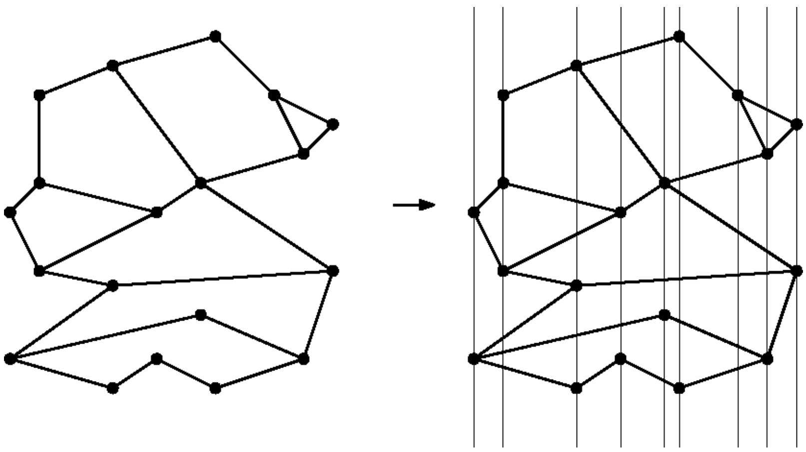


图6-2 条形分割

如 图 6-2 所示，在子区域划分的每个顶点处引入一条垂线。于是，整个平面就被分割成若干垂直的条带（slab）。我们将所有顶点按照x-坐标排序，并存入一个数组。这样，只需$O(\log n)$时间，就可以确定待查询点（query point）q所处的那根条带。在这根条带的内部，没有S的任何顶点。也就是说，我们的查找范围，已经从原来的整个子区域划分，缩小到该条带形区域的内部。每一条带都具有一种特殊的性质：与该条带 [①]相交的每一条边，都必然完全跨越它——因为，它们的端点不可能落在条带的内部。这就意味着，可以自上而下，对跨越该条带的所有直线进行排序。



图6-3 在任一条带内部，所有直线可以自上而下地明确排序

① 根据上下文，这里的条带形区域，应该除去做为其边界的（一或两条）直线。也就是说，它是开的。——译者

我们注意到，介于（排序后的）任何两条相邻边之间的区域，都属于 $\mathcal{S}$ 中的同一张面。在该条带中，最低和最高的两个区域都是无界的，而且同属于 $\mathcal{S}$ 的无界面。既然与某根条带相交的所有边（的相对位置）具有这样一种特殊的结构，我们就能够将它们存储为一个有序的数组。对其中的每一条边，我们都要做一个标记，指明在条带区域内，紧邻于其上方的是 $\mathcal{S}$ 的哪一张面。

至此已可以给出如下查询算法。首先，根据待查询点 q 的 x-坐标，在记录子区域划分中所有顶点 x-坐标的那个数组中，做一次二分查找，找到 q 所处的那根条带。然后，在该条带所对应的数组中，再次针对 q 进行第二次二分查找。其基本操作是：给定线段 s 以及点 q（通过 q 的垂线与 s 相交），要求判断出 q 究竟是位于 s 的上方、下方还是正好落在 s 上。如此便可确定在 q 下方、位置最高的那条线段（如果 q 的下方至少存在一条线段的话）。而通过该线段上的标记，就可以进一步找到 q 在 $\mathcal{S}$ 中所处的那张面。要是在 q 的下方根本就没有线段，则 q 必属于 $\mathcal{S}$ 的那张无界面。

该数据结构的查询时间性能很好——每次只需两次二分查找：第一次，数组的长度不超过 2n（在任一子区域划分中，n 条边至多对应于 2n 个端点）；第二次，数组的长度不超过 n（任何一根条带，最多与全部的 n 条边相交）。因此，查询时间为 $O(\log n)$。

那么，需要多大的存储空间呢？首先，需要一个数组来存放所有顶点的 x-坐标，这需要占用 $O(n)$ 空间。此外，每根条带还分别需要一个数组，用来存放与之相交（跨越）的所有边，每个这样的数组需要 $O(n)$空间。考虑到共有 $O(n)$根条带，这些数组总共需要占用 $O(n^2)$空间。

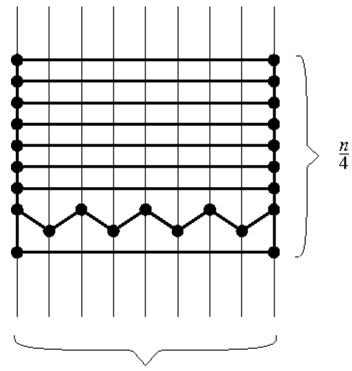


图6-4 条带划分的最坏情况

图 6-4 就给出了这样的一个子区域划分实例：其中包含$\frac{n}{4}$根条带，每根都分别与$\frac{n}{4}$条边相交——由此说明，上面（关于空间复杂度）的最坏估计，并非杞人忧天。

鉴于其所需的存储空间如此之大，这种数据结构很难让人感兴趣——在大多数实际应用中，规模为平方量级的结构，几乎等于毫无用处；即使是对中等大小的n而言，也是如此。（有的读者也许能够证明，该结构的空间复杂度的最坏情况，在实际中并不会出现。但是，存储量达到$O(n \times \sqrt{n})$的情况，还是很有可能出现的。）那么，导致平方量级存储量的根源何在？让我们再来看看 图 6-2。

实际上，原来的边以及通过各顶点的垂线，定义了一个新的子区域划分$\mathcal{S}'$。这个子区域划分中的每一张面，都是梯形（trapezoid）、三角形或者类似于梯形的无界面。而且，$\mathcal{S}'$是对原先的子区域划分的一个细分（refinement）——也就是说，$\mathcal{S}'$的任何一张面，都完全落在$\mathcal{S}$的某张面中。而实际上，上面所介绍的算法，就是在细分之后的这个子区域划分中进行点定位查询。（对$\mathcal{S}'$的查询结果）当然可以用来回答原先（针对$\mathcal{S}$）的平面点定位查询——既然$\mathcal{S}'$是$\mathcal{S}$的细分，所以只要能够在$\mathcal{S}'$中找到包含q的那张面，也就找到了包含q的$f \in \mathcal{S}$。然而不幸的是，经过细分之后，子区域划分的复杂度将高达平方量级。既然如此，对应的数据结构达到平方量级的规模，也就不足为怪了。

或许，我们应该寻找 $\mathcal{S}$ 的另一种细分方法。首先，这种细分应该与上面所介绍的分解方法一样——能够使点定位查询更易解答；但同时也应与上述分解方法不同——虽然其空间复杂度要比最初的子区域划分 $\mathcal{S}$ 稍高，却不会很高。实际上，这样的细分方法的确存在。本节的后续部分，将介绍所谓梯形图的概念——它正是我们所希望的一种细分方法。

若平面上两条线段的交为空，或者只相交于它们的一个共同端点，我们都称它们是互不相交的（non-crossing）。按照这一定义，在平面的任何一个子区域划分中，所有边都互不相交。

对于平面上互不相交的任意 n 条线段 S，都可以定义出与 S 相对应的一幅梯形图。不过，为了方便本节以及下一节的讨论，我们要 S 做两点简化假设。

首先，在场景的边缘会出现一些类似于梯形的无界面，为方便起见，我们要消除这些面。为此，可以引入一个（其边）与坐标轴平行的矩形 R，R 必须大到足以容纳下整个场景——也就是要能够容纳下 S 中的所有线段。这里讨论的是针对子区域划分的点定位查询，对这类问题来说，这个条件是可以满足的——在引入 R 之后，若待查询点落在 R 之外，则必然落在 S 中的那张无界面内。因此，尽管我们的注意力只集中在 R 的内部，也不会有任何问题。

第二项简化假设是：在S内所有线段的端点中，任何两个端点的x-坐标都是互异的。这一假设的理由，不象前一项那么容易说明。它的一个推论是，S中没有垂直边。这一假设与实际情况不甚相符——在很多应用中，垂直线段出现的频率很高；另外，由于给定坐标的精度往往是有限的，“分别来自互不相交的线段的两个端点的x-坐标相同”的情况，也并非难得一见。尽管如此，我们还是要做这一假设；在后面的第 6.3 节中，我们将对一般情况的处理进行讨论。

这样，我们所处理的对象，就是由n条线段构成的集合S，其中的线段互不相交，都包含在一个包围框（bounding box）R中，而且任何两个端点都不处于同一条垂线上。这样的集合，称为一般性位置线段集（a set of line segments in general position）。所谓S的梯形图$\mathcal{T}$(S)，也称作S的垂直分解（vertical decomposition）或者梯形分解（trapezoidal decomposition）。梯形图是这样得到的：经过S中每条线段的左、右端点，向上方和下方各发出一条垂直射线；在碰到S中的另一条线段或R的边界后，射线终止。由p发出的这两条垂直延长线，分别称为p的上垂直延长线（upper vertical extension）和下垂直延长线（lower vertical extension）。所谓S的梯形图，就是由线段集S、 矩形R以及所有上

垂直延长线和下垂直延长线导出的一个子区域划分。图 6-5 给出了这样的一个实例。

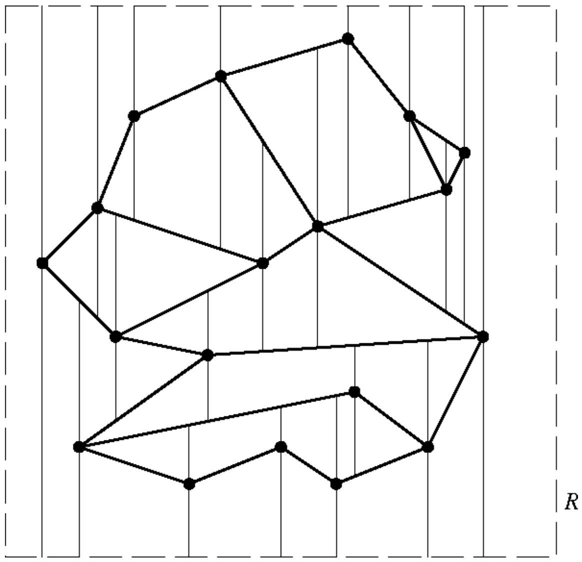


图6-5　点定位和梯形图

$\mathcal{T}(S)$中的每张面，都是由$\mathcal{T}(S)$的若干条边围成的。其中有些边可能是相邻的，甚至是共线的——如 图 6-6 所示，我们将（相对于某张面而言的）这类边合起来称作该面的一条侧边（side）。也就是说，所谓某张面的一条侧边，就是沿着该面的边界、长度极大化的一（直线）段。

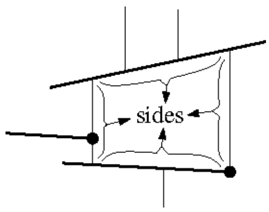


图6-6　梯形Δ的侧边

〖引理 6.1〗

任意给定一个一般性位置线段集 S，在 S 的梯形图中，每张面都有一到两条垂直的侧边，同时有且仅有两条非垂直的侧边。

〖证明〗

任取$\mathcal{T}(S)$中的一张面 f。首先来证明，f 是凸的。

既然 S 中的线段互不相交，则 f 的每个隅（corner）只有三种可能：或者是 S 中某条线段的端点，或者是某条垂直延长线与 S 的一条线段或 R 的一条边的接合位置，或者就是 R 的一个隅。若是第一种情况，则由于垂直延长线的引入，那里的内角不会超过 180°。若是第二种情况，则在该隅处的任何一个角都小于或等于 180°。而若是第三种情况，显然 R 的每个隅都是 90°。

总之，f 必是凸的——之所以所有的非凸性都能够被消除掉，是因为引入了垂直延长线。

我们关注的只是f的侧边，而不是T(S)的边，因此，既然f是凸的，则每张面至多有两条垂直的侧边。现在再来考虑非垂直的侧边。假设与引理的断言相反，f的非垂直侧边多于两条。若果真如此，则在这些非垂直侧边中，必有两条是前后相联的。而且，这两条侧边要么都属于f的下边界，要么都属于f的上边界。我们注意到：任何一条非垂直的侧边，要么是S中某条线段上的一段，要么就是R的某条边上的一段；此外，这些线段（以及边）互不相交。因此，上面那两条前后相联的侧边，其接合处必然是某条线段的一个端点。然而（按照梯形图的定义），这个端点也必然会发出两条垂直延长线——于是，这两条侧边就不可能是前后相联的[1]，与假设不合。因此，f至多只有两条非垂直的侧边[2]。

最后，我们注意到，f 是有界的（因为，整个场景的范围已经限制在一个包围框 R 之内了）。这就说明：它的非垂直侧边也不会少于两条，而垂直侧边则至少有一条。 ❑

〖引理 6.1〗指出，所谓梯形图的确名副其实——其中的每一张面要么是梯形，要么是三角形（也可以把三角形看成是梯形的特例，其中有一条边退化为一个点）。

从〖引理 6.1〗的证明中可以看出，梯形的任何一条非垂直侧边，要么是S中某条线段上的一段，要么是R的两条水平边之一上的一段。如 图 6-7 所示，任一梯形Δ的上方和下方，分别由（S的）一条非垂直线段或者（R的）一条水平边界定——它们分别记作top(Δ)和bottom(Δ)。

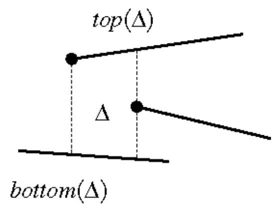


图6-7 梯形Δ的顶边与底边

若做了一般性位置假设（general position assumption），则任何梯形的垂直侧边要么是一条垂直延长线，要么是由两条垂直延长线接合而成的，要么就是 R 的两条垂直边之一。更准确地说，可以将梯形Δ的左、右侧边各划分成五类。比如，左侧边的五种可能是：

(a) 退化为一个点——也就是 top(Δ)和 bottom(Δ)共同的左端点；

(b) 由 top(Δ)的左端点向下发出的一条垂直延长线，直到与 bottom(Δ)相交；

(c) 由 bottom(Δ)的左端点向上发出的一条垂直延长线，直到与 top(Δ)相交；

---

[1] 更确切的表述应该是：于是，这两条侧边就不可能属于同一张面的边界。——译者

[2] 而且，这两条侧边必然分属 f 的上、下边界。——译者

(d) 由另一条线段 s 的右端点向上、下各发出一条垂直延长线，直到分别与 top(Δ)和 bottom(Δ)相交；

(e) R 的左边。在 $\mathcal{T}$(S)内的所有梯形中，只有一个是属于这种情况。具体讲，也就是 $\mathcal{T}$(S)中位置最靠左的那个梯形，这个梯形是唯一存在的。

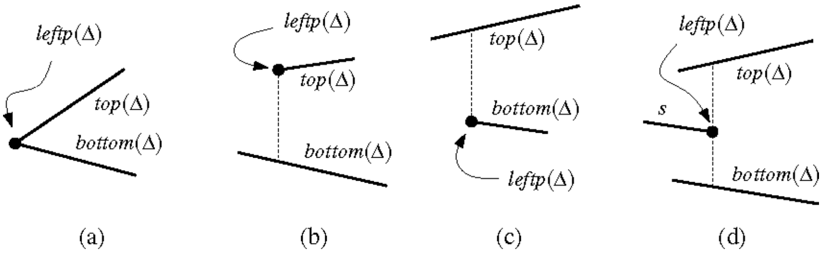


图6-8 梯形Δ左侧边共有5种可能，这里是其中的4种

图 6-8 给出了其中的前四种可能。对称地，Δ的右侧边也有五种可能。你可以自行验证一下，除上述五种情况之外，是否还有其它的可能。

任一梯形Δ ∈ $\mathcal{T}$(S)，只要它不是最左侧的那个，则在某种意义上，Δ的左侧垂直边是由某条线段的一个端点 p 确定的——这条垂直边属于由 p 发出的（两条）垂直延长线的一部分（当然，在退化情况下，这条垂直“边”就是 p 自己）。确定Δ左侧边的这个（线段）端点 p，记作 leftp(Δ)。正如在上面所看到的，leftp(Δ)要么是 top(Δ)或者 bottom(Δ)的左端点，要么就是另一条线段的右端点。对于唯一的那个以 R 的左边为左侧边的梯形Δ，leftp(Δ)被定义为“R 左下角的那个顶点”。类似地，确定Δ右侧边的那个（线段）端点 p，被记作 rightp(Δ)。请注意，Δ是由 top(Δ)、bottom(Δ)、leftp(Δ)和 rightp(Δ)唯一确定的。因此有时我们也会说，Δ是由这些线段和端点确定的。

任一子区域划分中各边所对应的梯形图，就是对该子区域划分的一个细分。那么，为什么说在梯形图中进行点定位，要比在一般的子区域划分中更容易呢？这一点并非一目了然。下一节将解释其原因。不过，在这里还是让我们首先来证实一下，梯形图本身的复杂度，并不会比定义它的原始集合中所包含的线段数目多多少。

〖引理 6.2〗

由任意 n 条处于一般性位置的线段组成的集合 S，其梯形图 $\mathcal{T}$(S)至多含有 6n + 4 个顶点，至多含有 3n+1 个梯形。

〖证明〗

$\mathcal{T}$(S)中的每个顶点，要么是 R 的某个顶点，要么是 S 中某条线段的端点，要么就是发自某个端点的一条垂直延长线与另一条线段（或者 R 的边界）之间的交点。既然每条线段的各个端点都会（向上、向下分别）引出两条垂直延长线，故顶点的总数就不会超过 $4 + 2n + 2 \times 2n$ =

6n + 4。

根据顶点总数的上界（upper bound），再利用欧拉公式，就可以得出其中所含梯形总数的上界。不过在这里将另辟蹊径，给出一个直接的证明。这个证明要借助 leftp(Δ)点。你应该记得，每个梯形Δ都对应于这样的一个点 leftp(Δ)。这个点要么是（n 条线段中的）某条线段的一个端点，要么就是 R 的左下角。只要对梯形左侧边的五种情况逐一检查一遍，就会发现：R 的左下角的确会为某个梯形担当这个角色，但是这样的梯形有且仅有一个；每条线段的右端点也只可能对至多一个梯形担当这个角色；而每条线段的左端点也最多只能担当两个不同梯形的 leftp(Δ)。（因为可能有多个端点相互重合，所以平面上的一个点可能同时担当多个梯形的 leftp(Δ)。然而，在情况(a)中只要我们约定"leftp(Δ)是 bottom(Δ)的左端点"，则任何一条线段 s 的左端点就至多只可能担当（分别位于 s 上方、下方的）两个梯形的 leftp(Δ)。）由此可知，梯形的总数不会超过 3n+1。 ❑

如果两个梯形Δ和Δ'接合于某条垂直边的左右，我们就称它们是相邻的（adjacent）。比如在 图 6-9(i)中，梯形Δ与Δ_1、Δ_2和Δ_3相邻，却与Δ_4和Δ_5不相邻。鉴于集合中的所有线段都处于一般性位置，故每个梯形至多与其它的四个（左、右各两个） 梯形相邻。若一般性位置假设不满足，则某个梯形有可能与任意多个梯形相邻（如 图 6-9(ii)所示）。

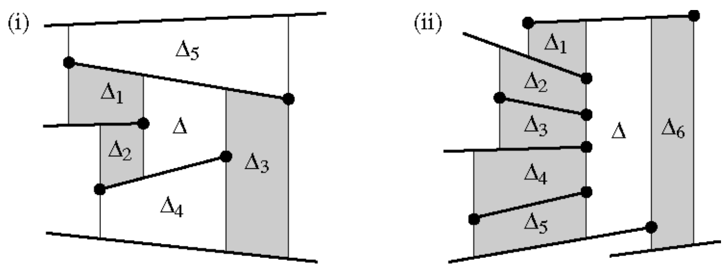


图6-9 与Δ相邻的梯形（用阴影填充）

考虑与Δ相邻于其左侧垂直边的某个梯形，记该梯形为Δ'。于是，要么top(Δ) = top(Δ')，要么bottom(Δ) = bottom(Δ')。若是前一种情况，我们称Δ'为Δ的"左上方邻居"（upper left neighbor）；若是后一种情况，则称Δ'为Δ的"左下方邻居"（lower left neighbor）。比如，按照这种定义，图 6-8(b)中的那个梯形有一个左下方邻居，但没有左上方邻居；图 6-8 (d)中的那个梯形既有一个左上方邻居，也有一个左下方邻居；图 6-8(a)中的那个梯形既没有左上方邻居，也没有左下方邻居（在所有的梯形中，以R的左侧边为左垂直边的那个唯一的梯形，也属于这种情况）。类似地，我们也可以定义出各梯形的右上方邻居（upper right neighbor）和右下方邻居（lower right neighbor）。

既然梯形图本身也是一种平面子区域划分，当然也可以借助第 2 章所介绍的双向链接边表（doubly-connected edge list）结构，来表示和存储梯形图。然而，鉴于梯形图的特殊结构，我们将使用另一种专门的数据结构，该结构在处理梯形图时会更加方便。这种数据结构利用了各梯形之间

的相邻关系，将整个子区域划分联接成为一个整体。对应于S中的每条线段、每个端点，在该结构中都要设置一个相应的记录——因为这些端点和线段可能会担当（某个梯形的）leftp(Δ)、rightp(Δ)、top(Δ)或bottom(Δ)。此外，对应于$\mathcal{T}$(S)中的每个梯形，在该数据结构中也要相应地设置一个记录；然而，无论是对于$\mathcal{T}$(S)中的边还是顶点，都不必设置相应的记录。对应于梯形Δ的记录配有若干指针，分别指向leftp(Δ)、rightp(Δ)、top(Δ)和bottom(Δ)，以及该梯形的（总共不超过四个）邻居。请注意，（按照这种存储方法，）每个梯形Δ的几何信息（即梯形各顶点的坐标）并不能显式地直接获得。尽管如此，一旦给定了leftp(Δ)、rightp(Δ)、top(Δ)和bottom(Δ)，梯形Δ就必然是唯一确定的。由此可知，根据（在该数据结构中）存储的有关Δ的信息，完全可以推导出该梯形的所有几何信息。

6.2 随机增量式算法

本节将建立一个随机增量式算法（randomized incremental algorithm），利用它可以为任意一组共 n 条（处于一般性位置的）线段 S，构造出对应的梯形图 $\mathcal{T}$(S)。在构造梯形图的过程中，该算法还会同时构造一个数据结构 $\mathcal{D}$，借助于它可以在 $\mathcal{T}$(S)中进行点定位查询。之所以没有采用平面扫描算法来构造梯形图，也正是出于这一考虑——诚然，平面扫描算法的确可以构造出梯形图，但是它却无法同时给出某种数据结构，以支持点定位查询。而完成后一任务，才是本章的主要目的。

在开始讨论算法之前，我们首先要对该算法所构造的数据结构 $\mathcal{D}$ 做一介绍。这种支持点定位查询的数据结构，被称作查找结构（search structure）。它是一幅有向无环图（directed acyclic graph - DAG），其中有唯一的根节点，同时对应于 S 的梯形图中的每个梯形，有且仅有一匹叶子。每个内部节点的出度都是 2。所有内部节点分为两类：x-节点和 y-节点。每个 x-节点都被标记为 S 中某条线段的一个端点；而每个 y-节点都被标记为某条线段。

在对点q进行查询时，要从根节点出发，沿着某条有向路径到达某匹叶子。最终到达的那匹叶子，就对应于$\mathcal{T}$(S)中包含q的那个梯形Δ。在沿查找路径前进时，每遇到一个新的节点，都要将其与q进行对比，以确定应该继续前进到其两个孩子节点中的哪一个。若是x-节点，则按如下形式进行比较：“取出存储于该节点处的那个端点；相对于经过该端点的那条垂线，q究竟是位于左侧还是右侧？”若是y-节点，则比较的方法如下：“取出存储于该节点处的那条线段s；相对于这条线段，q究竟是位于上方还是下方？”我们要保证，无论是遇到哪个y-节点，穿过q的那条垂线必然与该节点所存储的那条线段相交——唯此，（目的在于判断上方、下方的）比较才会有意义。在每个内部节点处，经过上述比较得出的结论无外乎两类：处于某个x-节点的左侧或者右侧；处于某个y-节点的上方或者下方。那么，要是待查询点正好落在某条垂线或者某条线段上，又应该如何处置呢？就目前而言，暂且假定这类情况不会发生。对于不满足一般性位置假设的线段集，应该如何处置？第 6.3 节将专门讨论这一问题，其中也涉及了这类待查询点的退化情况。

本算法所构造的查找结构 $\mathcal{D}$ 和梯形图 $\mathcal{T}$(S)相互关联：$\mathcal{T}$(S)中的梯形与 $\mathcal{D}$ 中的叶子一一对应；每一梯形Δ都可通过指针找到与之对应的叶子；反之，每匹叶子也可通过指针找到与之对应的梯形。

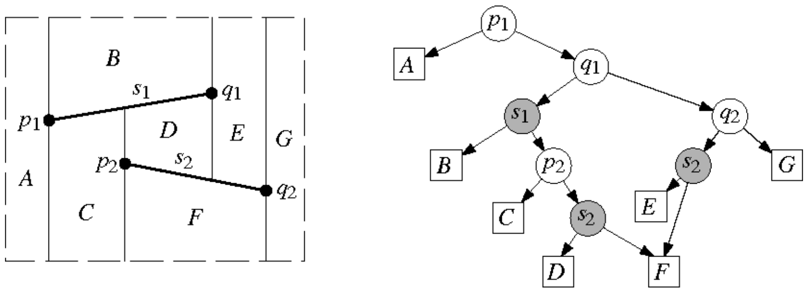


图6-10 两条线段的梯形图，及其对应的查找结构

在 图 6-10 中，画出了与两条线段s_1和s_2相对应的一幅梯形图，以及与该梯形图对应的查找结构。其中，白色的是x-节点，分别标有与之对应的线段端点；灰色的是y-节点，分别标有与之对应的线段；方块为查找结构的叶子，分别标有梯形图中与之对应的梯形。

这里将给出一个构造查找结构的递增式算法——也就是说，逐一引入各条线段；每增加一条线段，都相应地更新查找结构及对应的梯形图。各线段引入的次序，对最终生成的查找结构会有影响——按某些次序构造出来的查找结构，查询时间性能将很好；而按其它次序，时间性能却可能很差。为找到合适的次序，不必绞尽脑汁，只需再次沿用第 4 章（在解决线性规划问题时）所采用过的方法——随机化（randomization）。因此更确切地说，我们采用的实际上是一个随机增量式算法。稍后将证明：由这一随机增量式算法构造的查找结构，（时间）性能的期望值很好。不过，我们还是首先来具体介绍一下该算法。我们将从它的整体结构入手，然后再讲解它的各个子步骤。

算法 TRAPEZOIDALMAP(S)

输入：一组共 n 条互不相交的线段

输出：梯形图 $\mathcal{T}$(S)，以及与之对应的、限制于一个包围框之内的查找结构 $\mathcal{D}$

1. 构造一个包围框 R，其大小必须足以容纳 S 中的所有线段
   根据 R，初始化相应的梯形图结构 $\mathcal{T}$ 以及查找结构 $\mathcal{D}$
2. 将 S 中的所有线段随意打乱，得到一个随机序列：$s_1, s_2, \cdots, s_n$
3. **for** i ← 1 **to** n
4.   **do** 在（当前的）$\mathcal{T}$中，找到与s_i真相交[①]的所有梯形$\Delta_0, \Delta_1, \cdots, \Delta_k$
5.   将$\Delta_0, \Delta_1, \cdots, \Delta_k$从 $\mathcal{T}$ 中删去
           将它们替换为由于 s_i 的引入而新生出来的若干梯形

---

① 真相交，指线段与梯形相交于内部。也就是说，不考虑线段的端点与梯形的边界。——译者

6.	将与$\Delta_0, \Delta_1, \cdots, \Delta_k$对应的叶子从 𝒟 中删去 对应于每个新生成的梯形，生成一匹新的叶子 将新生出的叶子与已有的内部节点相联接 (* 正如下面要介绍的，为此可能需要加入一些新的内部节点 *)

接下来要介绍该算法各个步骤的详细实现。在后续的讲解中，令 $S_i := \{s_1, s_2, \cdots, s_i\}$。TRAPEZOIDALMAP 算法中的循环部分具有如下不变性：每经过一次循环，当前的 𝒯 就是对应于 S_i 的梯形图，而当前的 𝒟 则是与 𝒯 对应的一个正确的查找结构。

第一行是对𝒯和𝒟做初始化，$\mathcal{T}(S_0) = \mathcal{T}(\varnothing)$。不难看出，经这一步之后，上述不变性的确满足：与空集对应的梯形图，只包含一个梯形——也就是包围框R本身；而与$\mathcal{T}(S_0)$相对应的查找结构只有（对应于R的）一匹叶子。有关随机排列的生成方法，请参见第 4 章。最后来考察一下，在第 4~6 行中应该如何插入一条新的线段s_i。

为了对当前的梯形图进行更新，首先必须了解清楚，（在引入 s_i之后）会发生哪些变化。实际上，只有与 s_i 相交的那些梯形，才会有所变化。更准确地说，$\mathcal{T}(S_{i-1})$中的某个梯形不会在$\mathcal{T}(S_i)$中出现，当且仅当 s_i 与该梯形相交。因此首先要做的，就是找出这些梯形。

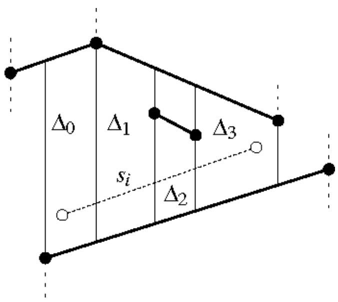


图6-11 与s_i相交的梯形

如 图 6-11 所示，按其与s_i相交的次序，将这些梯形从左到右记作$\Delta_0, \Delta_1, \cdots, \Delta_k$。我们注意到，$\Delta_{j+1}$ 必然是Δ_j的右邻居之一。具体是哪个邻居，通过比较不难判断：若rightp(Δ_j)位于s_i的上方，则Δ_{j+1} 必是Δ_j的右下方邻居；否则，就是Δ_j的右上方邻居。这样，只要找到了Δ_0，就可以通过对梯形图结构的遍历（traversal），顺藤摸瓜地找出$\Delta_1, \cdots, \Delta_k$。因此，在进行遍历之前，需要在𝒯中进行查找，以确定s_i的左端点p所在的那个梯形Δ_0。根据此前所做的一般性位置假设，只要p不是作为一个端点出现在S_{i-1}中，它就必然落在Δ_0 的内部。这说明，只要在$\mathcal{T}(S_{i-1})$中对p进行一次点定位查询，就可以找到Δ_0。这样，（该算法中）最令人兴奋的一点就出现了：既然在算法的这个阶段，𝒟就是$\mathcal{T} = \mathcal{T}(S_{i-1})$所对应的查找结构，所以我们所要做的，无非就是在𝒟上对点p进行一次查询。

若在 S_{i-1} 中，p 已经作为一个端点存在了（应该记得，我们允许多条线段拥有共同端点），则需格外仔细。为找到Δ_0，首先要对 𝒟 进行查找。若 p 碰巧尚未出现，则查找算法可正常运行，不会遇

到任何麻烦，最终必能找出对应于Δ_0的那匹叶子。但若 p 已经存在，则会出现以下问题：在查找过程中，某个 x-节点的对应端点与 p 落在同一条垂线上。应该记得，这类待查询点是当作非法的。为补救这一问题，我们假想着将 p 替换为其右侧与之相距很近的一个点 p'，然后继续查找下去。在实现这一查找过程时，这种做法实际的效果可以理解为：一旦 p 落在某个 x-节点所在的垂线上，就把它当成是位于该垂线的右侧。类似地，要是 p 正好落在某个 y-节点所对应的线段 s 上（这种情况的出现只有一种可能——点 p 既是 s_i 的左端点，也是 s 的左端点），就要比较 s 与 s_i 的斜率——若 s_i 的斜率更大，则认为 p 位于 s 的上方；否则，就认为 p 位于下方。经过这些调整，才能保证查找能够顺利进行，并最终找出与 s_i 真相交的第一个梯形Δ_0。总而言之，可以通过下述算法，找出$\Delta_0, \cdots, \Delta_k$。

```
算法 FOLLOWSEGMENT(T, D, si)
输入：梯形图 T，与 T 相对应的查找结构 D，以及新近引入的一条线段 si
输出：由所有与 si 真相交的梯形（自左向右）组成的一个序列：Δ0, …, Δk
1.   分别令 p 和 q 为 si 的左、右端点
2.   在查找结构 D 中对 p 进行查找，最终找到梯形Δ0
3.   j ← 0
4.   while (q 位于 rightp(Δj)的右侧)
5.       do if (rightp(Δj)位于 si 的上方)
6.              then 令Δj+1 为Δj 的右下方邻居
7.              else  令Δj+1 为Δj 的右上方邻居
8.          j ← j + 1
9.   return (Δ0, Δ1,⋯, Δj)
```

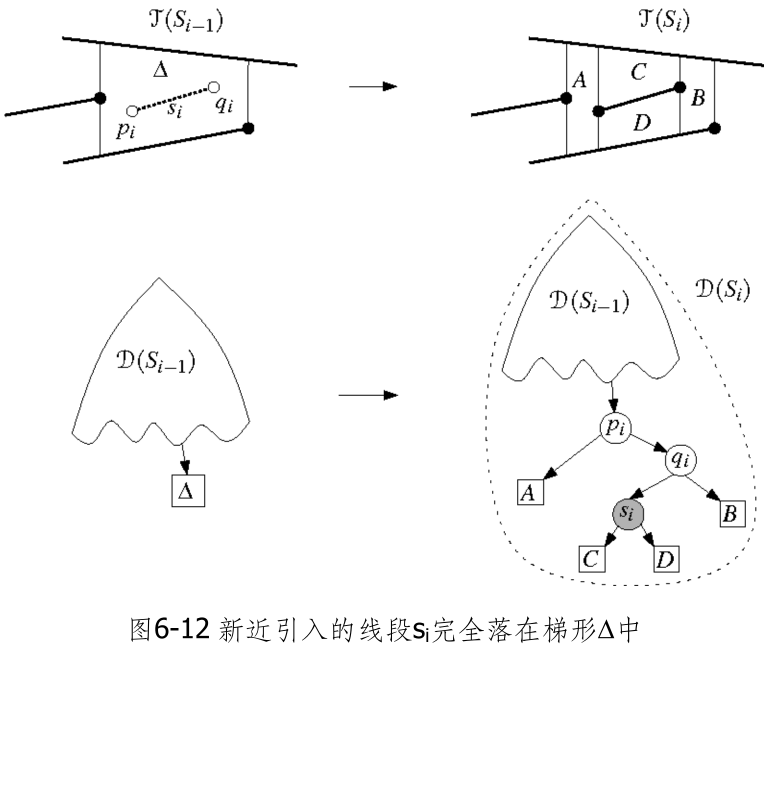


图6-12 新近引入的线段s_i完全落在梯形Δ中

以上告诉我们，应如何找出与s_i相交的所有梯形。接下来还要对$\mathcal{T}$和$\mathcal{D}$做相应的更新。我们从一种简单的情况入手，即：s_i完全落在某个梯形$\Delta = \Delta_0$之中。这就是 图 6-12 的左边所描绘的情况。

为了更新$\mathcal{T}$，我们要先将Δ从$\mathcal{T}$中删除掉，然后将其替换为四个新的梯形 A、B、C 和 D。请注意，我们已经拥有丰富的信息，足以对这些新梯形各自对应的记录（各梯形的邻居、其顶边与底边所在的线段以及确定其左、右侧边的端点）进行正确的初始化。实际上，根据线段 s_i 以及存储的有关Δ的信息，记录中的这些域的值都可以在常数时间内确定。

最后要对$\mathcal{D}$进行更新。我们为此所需要做的工作，仅仅是将原先对应于Δ的那匹叶子，替换为一棵包含四匹叶子的小树。这棵树中含有两个x-节点，分别用来对s_i的左、右端点进行比较；还有一个y-节点，用来对线段s_i本身进行比较。只要已经确定了待查询点落在Δ内，凭借这些结构就足以进一步判断出，该待查询点到底落在这四个新梯形A、B、C和D中的哪个之中。图 6-12 的右边所显示的，就是更新后的查找结构。请注意，线段s_i的左端点有可能与leftp(Δ)重合，其右端点也可能与rightp(Δ)重合，这两种情况甚至还可能同时发生。果真发生这种情况，新生出的梯形就只有三个或者两个。尽管如此，我们依然可以按照同样的原则，完成相应的更新。

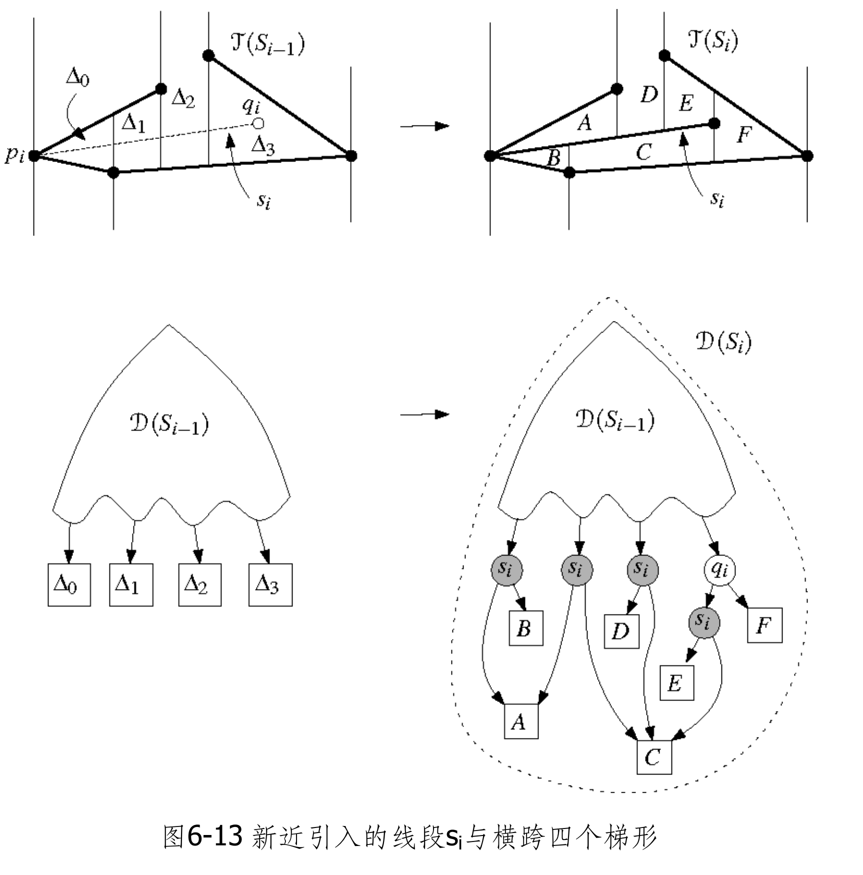


图6-13 新近引入的线段s_i与横跨四个梯形

与上述最简单的情况相比，s_i与两个甚至更多个梯形相交的情况，只是略微复杂一些。将所有与s_i相交的梯形组成序列$\Delta_0, \Delta_1, \cdots, \Delta_k$。为更新$\mathcal{T}$，首先要从$s_i$的两个端点处发出垂直延长线，分别将$\Delta_0$和$\Delta_k$一分为三（倘若$s_i$的端点已经在$\mathcal{T}$中出现，这一步处理就不必进行）。然后，要将当前与$s_i$相交

的各条垂直延长线截短。其效果如 图 6-13 所示：s_i附近的一些梯形将合并起来。借助存储在$\Delta_0, \Delta_1, \cdots, \Delta_k$中的信息，可在线性正比于（与$s_i$）相交梯形数目的时间内，完成这一步操作。

为更新$\mathcal{D}$，必须删除对应于$\Delta_0, \Delta_1, \cdots, \Delta_k$的叶子，然后为每个新的梯形分别生成一匹叶子，最后还要引入若干新的内部节点。更准确地说，处理的过程应该如下：若s_i的左端点落在Δ_0的内部（也就是说，梯形Δ_0已经因此被一分为三），则用一个对应于s_i左端点的x-节点，以及另一个对应于线段s_i本身的y-节点，来替换原先对应于Δ_0的那匹叶子。类似地，若s_i的右端点落在Δ_k的内部，则用一个对应于s_i右端点的x-节点，以及一个对应于线段s_i本身的y-节点，来替换原先对应于Δ_k的那匹叶子。最后，与$\{\Delta_1, \Delta_2, \ldots, \Delta_{k-1}\}$相对应的所有叶子，都将被替换为同一个y-节点，该节点对应于线段s_i。新生成的各内部节点的出弧，都需要正确地指向对应的新叶子。请注意，因为我们已经从$\mathcal{T}$中原来的某些梯形中划分出若干个子梯形，并对这些子梯形进行了合并，所以可能会有多条边，同时指向某个新引入的梯形。你可以从 图 6-13 中看到这一现象。

以上介绍的算法不仅可以构造出$\mathcal{T}(S)$，而且同时也建立起一个与之对应的查找结构$\mathcal{D}$。在给出算法 TRAPEZOIDALMAP 之后我们曾经指出，其中的循环具有一种不变性。根据这种不变性，该算法的正确性很容易得到证明。因此，最后要做的只有一件事——分析该算法的性能。

具体按照何种次序来处理各条线段，对最终生成的查找结构$\mathcal{D}$以及算法本身的运行时间，都有相当大的影响。有时，生成的查找结构会达到平方量级的规模，相应的查询时间是线性的；而在另外一些时候，尽管是同一线段集，但由于各线段的排列次序不同，所生成查找结构的性能可能会更好。与第 4 章的做法一样，这里并不准备刻意去找出某个最优的序列，而只是按照一个随机次序（逐一）插入各条线段。如此，就要根据概率来分析该算法的性能——也就是说，我们将要考察的，是该算法及其查找结构的期望性能（expected performance）。在目前的上下文中，“期望”一词的确切含义或许还不甚明确。任取某个固定的线段集S，其中包含n条互不相交的线段。现通过算法 TRAPEZOIDALMAP，构造出$\mathcal{T}(S)$所对应的查找结构$\mathcal{D}$。这个结构（的具体组成）取决于算法中第 2 行所确定的（线段）排列次序。n个不同对象可能的排列，共有n!种，因此，（对于这一固定的输入，）算法可能的处理过程有n!种。而所谓该算法的期望运行时间（expected running time），就是所有这n!种排列各自所对应的运行时间的平均值。另一方面，不同的排列次序，也将导致不同的查找结构。而所谓$\mathcal{D}$的“期望规模”（expected size），就是这n!个查找结构的平均规模。最后，所谓某个点q的“期望查询时间”（expected query time），指的是在这n!个查找结构中对q进行查找所需的平均时间。请注意，此处指的期望时间（expected time）只是对某个具体的点q而言，而不是泛泛地指某个查找结构（对于所有可能的待查询点q而言的）“最大平均查询时间”。要得到并证明针对后一指标的一个上界，需要更多的技巧，因此将押后到第 6.4 节再予回答。

〖定理 6.3〗

对于由处于一般性位置的任意 n 条线段构成的集合 S，算法 TRAPEZOIDALMAP 都可以在 $O(n\log n)$的期望时间内，计算出 S 的梯形图 $T(S)$以及与之对应的查找结构 D。该查找结构的期望规模为 $O(n)$；对任何待查询点 q，期望查询时间为 $O(\log n)$。

〖证明〗

正如此前所提到的，该算法的正确性，可以由其中的循环所具有的不变性直接得证，因此我们只需将注意力集中于对其性能的分析。

首先考虑查找结构 D 的查询时间。选取一个固定的待查询点 q。在对 q 进行查找的过程中，需要在 D 中依次访问若干节点，这些被访问到的节点构成了（从根节点通往叶子的）一条访问路径，而 q 所对应的查询时间，就线性正比于这条路径的长度。因此，只要找出查找路径（平均）长度的上界（，也就得到了期望查询时间的上界）。只要简单地对各种情况逐一分析，我们就可以知道，算法每经过一次迭代，D 的深度（即最大路径长度）的增加量不会超过 3。因此，q 对应的查询时间不会超过 3n。请注意，这只是考虑 S 所有可能的插入次序、在最坏情况下性能的一个最紧上界。然而在这里，我们所感兴趣的并不是最坏情况下的性能，而是期望性能——按照不同插入次序生成的全部 n!个查找结构，我们都要考虑到，并且统计出 q 在这些结构中查询时间的平均值，并进而得出平均查询时间的上界。

现考虑在 D 中查找 q 时所访问的那条路径。沿此路径的各个节点，都诞生于算法的某轮迭代过程。令 X_i 为第 i 轮迭代中该路径上新生成的节点数目，$1 \le i \le n$。既然在这里我们把 S 和 q 都看作是固定的，故 X_i 就是一个随机变量——它只取决于各线段的随机插入次序。现在，我们就可以将期望的路径长度表示为：

$$E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$$

这个等式的成立，是由期望的线性律（linearity of expectation）保证的——总和的期望值，等于各期望值的总和。

我们此前已经观察到：每经过一轮迭代，在任一待查询点所对应的查找路径上，最多增加 3 个节点。由此可知，$X_i \le 3$。也就是说，若将沿着 q 所对应的查找路径、在第 i 轮迭代中有一个新节点诞生的概率记作 P_i，则有：

$$E[X_i] \le 3P_i$$

为了得出 P_i 的上界，还需要利用到这样一个重要的观察结果：若第 i 轮迭代为 q 所对应的查找路径贡献一个节点，则 q 于 $T(S_{i-1})$中所在的那个梯形，必与 q 于 $T(S_i)$中所在的那个梯形不同——亦即，$\Delta_q(S_{i-1})$必与$\Delta_q(S_i)$有所不同。这就是说，

$$P_i = \Pr[\Delta_q(S_i) \neq \Delta_q(S_{i-1})]$$

若$\Delta_q(S_i)$果真与$\Delta_q(S_{i-1})$与有所不同，则$\Delta_q(S_i)$必然是诞生于第 i 轮迭代中的梯形之一。请注意，在第 i 轮迭代中生成的每一个梯形Δ，都必然与 s_i（即在该轮迭代中被插入的那条线段）相邻——要么 s_i 是 top(Δ)或者 bottom(Δ)，要么 s_i 的某个端点是 leftp(Δ)或者 rightp(Δ)。

现在，考虑某一固定子集$S_i \subset S$。做为S_i的函数，梯形图$\mathcal{T}(S_i)$是唯一确定的，进而$\Delta_q(S_i)$也是由S_i唯一确定的。这样，$\Delta_q(S_i)$就与S_i中各条线段的插入次序无关。由于s_i的插入，q所在的那个梯形可能会有所改变，但发生变化的可能性有多大呢？为界定这个概率，需要再次使用第 4 章曾采用过的技巧——后向分析（backward analysis）。考察$\mathcal{T}(S_i)$，并假想着反过来把线段s_i删除掉（并倒退回$\mathcal{T}(s_{i-1})$）。这样，$\Delta_q(S_i)$就有可能从梯形图（$\mathcal{T}(S_i)$）中消失掉。这种情况发生的概率有多大呢？由前面的分析可知：在移除s_i之后$\Delta_q(S_i)$随之消失，当且仅当top($\Delta_q(S_i)$)、bottom($\Delta_q(S_i)$)、leftp($\Delta_q(S_i)$)或rightp($\Delta_q(S_i)$)之一随之消失。其中，top($\Delta_q(S_i)$)消失的概率是多少呢？既然S_i中的线段是按照随机次序插入的，故S_i中每条线段为s_i的概率都是相等的。这就是说，s_i正好是top($\Delta_q(S_i)$)的概率应该等于$\frac{1}{i}$。（若top($\Delta_q(S_i)$)为整个场景包围框R的上缘，则此概率均为零。）类似地，s_i正好是bottom($\Delta_q(S_i)$)的概率也不会超过$\frac{1}{i}$。当然，leftp($\Delta_q(S_i)$)可能同时是多条线段的端点——此时，s_i是这些线段之一的概率将会很大。然而，若leftp($\Delta_q(S_i)$)消失了，则s_i必然是S_i中唯一的那条以leftp($\Delta_q(S_i)$)为端点的线段。因此，leftp($\Delta_q(S_i)$)消失的概率同样不会超过$\frac{1}{i}$。这个结论对于rightp($\Delta_q(S_i)$)也是适用的。总而言之，我们有：

$$P_i = \Pr[\Delta_q(S_i) \neq \Delta_q(S_{i-1})] = \Pr[\Delta_q(S_i) \notin \mathcal{T}(S_{i-1})] \leq \frac{4}{i}$$

（有一个小小的技术性问题：上述论证中固定了 S_i。这似乎意味着，我们所得到的关于 P_i 的上界之所以能够成立，必须以"S_i 是某个固定集合"为前提。尽管如此，因为我们得到的上界并不依赖于这个固定的集合具体是谁，所以实际上，该上界的成立并不需要任何前提条件。）

将上述结果归纳到一起，就得到了查询时间的一个上界：

$$E[\sum_{i=1}^{n} X_i] \leq \sum_{i=1}^{n} 3P_i \leq \sum_{i=1}^{n} \frac{12}{i} = 12\sum_{i=1}^{n} \frac{1}{i} = 12H_n$$

这里的 H_n，就是所谓调和数（harmonic number）的第 n 项。其定义为：

$$H_n := \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

在分析很多算法时，都会遇到这个调和数，因此，你最好能够记住调和数的上、下界：

$$\ln n < H_n < \ln n + 1, \quad (n > 1)$$

（只要将 H_n 与积分 $\int_1^n \frac{1}{x}dx = \ln n$ 做一比较，就可以得出以上的上、下界。）由此可得结论：对于某个固定的待查询点，期望查询时间为 $O(\log n)$——这正是本定理的结论之一。

接下来，考虑$\mathcal{D}$的规模。要想界定这个结构的规模，只需界定出$\mathcal{D}$中所含节点的数目。我们首先注意到，$\mathcal{D}$中的各匹叶子，与Δ中的各个梯形一一对应。根据〖引理 6.2〗，其数目为$O(n)$。这就说明，所有节点（包括叶子及内部节点）的总数不会超过

$$O(n) \quad + \quad \sum_{i=1}^{n}(\text{第}i\text{轮迭代中生成的内部节点数目})$$

令 k_i 为在第 i 轮迭代中（由于线段 s_i 的引入而）新生成梯形的数目。换而言之，k_i 就是 $\mathcal{D}$ 中新生成叶子的数目。当然，第 i 轮迭代中新生成的内部节点有 k_i-1 个。在$\mathcal{T}(S_i)$中，新生成的梯形显然不可能比其中所有梯形的总数还多。只要注意到这一点，就可以马上得出 k_i 的最坏情况下的一个上界——$O(i)$。由此，可以进一步得出关于整个结构规模的一个最坏情况上界：

$$O(n) + \sum_{i=1}^{n} O(i) \quad = \quad O(n^2)$$

事实上，要是运气不佳，以至于我们插入各线段的次序糟糕透顶，构造出来的 $\mathcal{D}$ 的确会达到平方量级的规模。不过在这里，我们更感兴趣的，还是该数据结构（对所有可能的插入次序进行平均之后）的期望规模。再次根据期望的线性律，我们得出该期望规模的上界为：

$$O(n) + E[\sum_{i=1}^{n}(k_i\text{-}1)] \quad = \quad O(n) + \sum_{i=1}^{n}(E[k_i])$$

这样，我们就只需界定出 k_i 的大小范围。在此前导出查询时间上界的过程中，我们已经为此准备好了各种必要的工具。同样地，考虑某个固定的集合 $S_i \subset S$。对任一梯形$\Delta \in \mathcal{T}(S_i)$以及任一线段 $s \in S_i$，令

$$\delta(\Delta, s) \quad := \quad \begin{cases} 1 & \text{（若在将}s\text{从}S_i\text{中删除之后，}\Delta\text{会随之消失）} \\ 0 & \text{（否则）} \end{cases}$$

在此前对查询时间进行分析时，我们已经注意到：一个梯形的消失，只可能由至多 4 条线段（的删除）而引起。因此，

$$\sum_{s \in S_i} \sum_{\Delta \in \mathcal{T}(S_i)} \delta(\Delta, s) \quad \le \quad 4 \times |\mathcal{T}(S_i)| \quad = \quad O(i)$$

此处的 k_i，就是在插入 s_i 之后随之而生的新梯形的数目；当然，反过来，k_i 也是在将 s_i 从$\mathcal{T}(S_i)$中删除之后，随之而亡的原有梯形的数目。既然 s_i 是来自 S_i 的一个随机元素，我们只要对

所有的 $s \in S_i$ 做一平均，就可以得出 k_i 的期望值：

$$E[k_i] = \frac{1}{i} \times \sum_{s \in S_i} \sum_{\Delta \in \mathcal{T}(S_i)} \delta(\Delta, s) \leq \frac{O(i)}{i} = O(1)$$

由此我们可以得出结论：在算法的每一轮循环中，新生成梯形的期望数目为 $O(1)$。这样，也就立即得出了（数据结构 $\mathcal{D}$ 需要占用）存储空间的期望规模——$O(n)$。

唯一尚待完成的任务，是确定该构造算法的运行时间上界。在对查询时间及存储空间做过分析之后，这项任务已经没有多大的难度。我们只需观察到：插入线段 s_i 所需的时间，等于 $O(k_i)$ 再加上在 $\mathcal{T}(S_{i-1})$ 中对 s_i 的左端点进行定位所需的时间。根据此前所得出的有关 k_i 以及查询时间的上界，马上就可以得出该算法的期望运行时间为：

$$O(1) + \sum_{i=1}^{n} \{ O(\log i) + O(E[k_i]) \} = O(n\mathbf{log}n)$$

证毕。 ❑

请再次注意，〖定理 6.3〗中所指的期望值，仅仅是针对算法能够做出的各种随机选择而言的平均值，而不是对所有可能的输入进行平均。因此，不存在坏的输入——也就是说，对由n条线段组成的任何输入，该算法的期望运行时间都是$O(n\mathbf{log}n)$。

如果考虑所有可能的待查询点，则最大期望查询时间（expected maximum query time）又会有多长呢？正如前面所提到的，〖定理 6.3〗并没有对此做出任何保证。不过，第 6.4 节将会证明：期望的最大查询时间同样也是$O(\mathbf{log}n)$。因此，我们可以构造出一个期望规模为$O(n)$的数据结构，借助这个结构，期望查询时间不会超过$O(\mathbf{log}n)$。同时这也说明：的确存在一种大小为$O(n)$、对任何待查询点的查询时间均为$O(\mathbf{log}n)$的数据结构（参见〖定理 6.8〗）。

最后回到自己最初的问题——在一个平面子区域划分S中进行点定位。我们假定，S的给出形式为由n条边组成的一个双向链接边表结构。针对与S中各边相对应的那幅梯形图，我们利用算法TRAPEZOIDALMAP构造出一个与之对应的查找结构$\mathcal{D}$。然而，为了将这一查找结构应用于在S中的点定位，我们还需要为$\mathcal{D}$中的每匹叶子配备一个指针，分别指向S中的某张面——$\mathcal{T}(S)$中与该叶子对应的那个梯形，就包含在这张面之内。不过，这并不是什么难事。你应该能够回忆起来，根据第 2 章的介绍，在对应于S的双向链接边表中，为每条半边（half-edge）都配备了一个指针，指向从左侧与之关联的那张面。另外，借助Twin($\vec{e}$)，可以在常数时间内找到从右侧与之关联的那张面。因此，对于$\mathcal{T}(S)$中的任何一个梯形Δ，只需在S中找到从下方与top(Δ)关联的那张面，然后对该面进行考察。若top(Δ)是R的顶边，则Δ必包含于S中唯一的那个无界面之中。

接下来的一节将指出："各线段必须处于一般性位置"这一假设，完全是不必要的。这样，就可以得出〖定理 6.3〗的一个新版本，该版本的限制条件更少。由此可以得出如下推论。

〖推论 6.4〗

给定由任意 n 条边构成的一个平面子区域划分 S。可在 O(nlog**n)期望时间内，构造出一个期望规模为 O(n)的数据结构；借助该结构，对任一待查询点的点定位查找，都可在 O(**log**n)期望时间内完成。**

6.3 退化情况的处理

前面数节一直做了两个简化性假设。首先假定了，集合中的各条线段都处于一般性位置——也就是说，其中任何两个端点的 x-坐标互不相同。每个待查询点，都对应于一条查找路径；沿着这条路径，要经过多个 x-节点和 y-节点。对此我们还假定，该待查询点与各 x-节点都不会落在同一条垂线之上，也不会正好落在与某个 y-节点相对应的线段上。现在，我们就来着手消除这些假定条件。

首先让来看看，如何消除"不同端点不会落在同一条垂线上"的假定。请注意：给定一组线段，其梯形图的确是由我们所选取的垂直方向确定的，然而（从支持点定位查找这一功能的角度来看），具体选取那个方向并不是本质性的问题——这一观察结果十分关键。因此，可以对坐标系略做旋转。只要旋转的角度足够小，就不会有任何两个不同的端点落在同一垂线上。然而，过小角度的旋转，又会在数值（计算）方面引发问题。为了保证（旋转）计算的正确性，必须能够进行极高精度的计算——即使输入的坐标本来就是整数，亦是如此。进行这种旋转的更好的方法，是所谓的"符号变换"（symbolic transformation）。在第 5 章中，我们曾经见到过符号变换的另一种形式——合成数。我们在那里利用过这种方法，来处理不同点的x-坐标或y-坐标相同的情况。本章将再次研究这种符号扰动（symbolic perturbation）的方法，并从几何的角度来理解它。

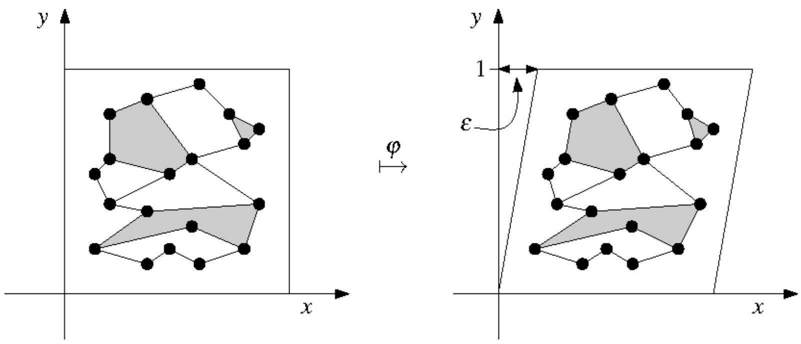


图6-14 剪切变换

实际进行的并非旋转，而是一种称作"剪切变换"（shear transformation）的仿射变换，它更加

方便。具体说，这里所采用的，是沿着 x-坐标方向、偏移量为$\varepsilon > 0$的一个剪切变换：

$$\varphi \ : \ \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x+\varepsilon y \\ y \end{pmatrix}$$

剪切变换的效果，如 图 6-14 所示。经过这种变换，所有垂线都将成为斜率为$\frac{1}{\varepsilon}$的直线。这样一来，原先落在同一条垂线上各点，现在的x-坐标必然互异。此外，给定任何一组输入点，只要$\varepsilon > 0$足够小，所有这些点沿x-方向的次序将依然保持原样。为了保证这一性质的成立，ε不能超过某一上界——计算出这一上界，并非难事。因此，以下讨论中将假定：总是能够找到某个足够小的$\varepsilon > 0$，使得在经过剪切变换之后，各输入点（沿x-方向）的次序保持不变。令人惊讶的是，稍后我们将会发现，根本就不必真正去计算出ε的具体数值。

设集合 S 由任意 n 条互不相交的线段组成，我们将对集合$\varphi S := \{\varphi s : s \in S\}$使用 TRAPEZOIDALMAP 算法。然而，正如此前所提到的，倘若真地去进行这一变换，将会引发数值计算方面的问题。因此，我们将使用如下技巧：将点$\varphi p = (x+\varepsilon y, y)$直接存储为$(x, y)$。这种表示方法是唯一的。而我们只需要确定，如果将以这种方式表示的一组线段交给该算法，算法依然可以正确地进行处理。该算法并不需要对任何的几何对象进行计算——比如，我们实际上并不需要计算出各垂直延长线的端点坐标。这个算法对输入点集所需进行计算，不外乎两类基本的操作。第一类操作是：任取不同的两个点 p 和 q，以通过 p 的垂线为基准，判断 q 究竟是位于其左侧、右侧，还是正好落在这条垂线上。第二类操作是：取出以 p_1 和 p_2 为端点的一条输入线段，判断第三个点 q 究竟是位于这条线段上方、下方，还是正好落在这条线段上。只有在我们首先确定了“经过 q 的垂线与该线段相交”之后，才会进而实施第二类操作。无论是 p、q 还是 p_1、p_2，都是输入集 S 中某条线段的端点。（读者可以重温此前对该算法的描述以验证一下，仅靠这两种操作就足以实现这个算法。）

首先来看看，在经过上述变换之后，如何将第一类操作应用于两个点φp和φq。这两个点的坐标分别为$(x_p+\varepsilon y_p, y_p)$和$(x_q+\varepsilon y_q, y_q)$。若 $x_q \neq x_p$，则通过 x_q 和 x_p 之间的（大小）关系，就足以确定测试的结果——我们在选取ε的时候，已经保证了这一性质。反之，若 $x_q = x_p$，则这两个点沿水平方向的次序，将取决于 y_q 和 y_p 之间的（大小）关系。总而言之，沿着水平方向，任何两个不同的点之间都有一个严格的次序。这样，除非 p 与 q 位置重合，φq 绝不可能与φp 落在同一条垂线上。任何两个不同的点，不得具有相同的 x-坐标——这正是我们所希望的。

第二类操作的输入为一条线段φs，其端点分别为$\varphi p_1 = (x_1+\varepsilon y_1, y_1)$和$\varphi p_2 = (x_1+\varepsilon y_2, y_2)$，我们的任务是通过比较判断出，$\varphi q = (x+\varepsilon y, y)$究竟是位于$\varphi s$ 的上方、下方，还是正好落在该线段上。每次进行这种比较时，算法都会保证，经过φq 的垂线必然与φs 相交。也就是说，

$$x_1 + \varepsilon y_1 \ \leq \ x + \varepsilon y \ \leq \ x_2 + \varepsilon y_2$$

这就意味着 $x_1 \le x \le x_2$。此外，若 $x = x_1$，则 $y \ge y_1$；若 $x = x_2$，则 $y \le y_2$。下面，将对这两种情况加以区分。

若 $x_1 = x_2$，则在变换之前线段 s 必是垂直的。此时，必有 $x_1 = x = x_2$，故 $y_1 \le y \le y_2$——也就是说，此时的 q 正好落在线段 s 上（即 q 与 s 相互关联）。经过仿射变换，关联性依然保持——亦即，在变换之前相互重合的两个点，经变换之后依然重合。由此可以得出结论，φq 依然落在φs 上。

再考虑 $x_1 < x_2$ 的情况。我们已经知道，φq 所在的垂线必与线段φs 相交，因此的确可以将点φq 与线段φs 进行比较。现在，我们可以进一步观察到，映射φ能够保持点与直线之间的相对次序——原先位于某条直线上方（下方，或者与之重合）的点，经变换之后，依然位于（经变换后的）该直线的上方（下方，或者与之重合）。因此，只需对原先未经变换的点 q 和线段 s 进行比较。

上述分析说明，若将φS（而不是 S）做为算法的输入，则需要进行的改动只有一处：应该按照字典序，来确定两个点沿水平方向的次序。显然，调整之后的算法依然能够构造出φS 的梯形图以及相应的查找结构 Ʈ(φS)。请注意，正如此前所承诺的那样，实际上并不需要知道ε的具体数值，因此，完全不必在算法运行之初确定它的大小。对我们有用的条件只有一个——ε必须足够小。

以上，借助剪切变换消除了对输入数据的第一个假定条件——不同端点不得具有相同的 x-坐标。还有另一个假定条件：待查询点不得与其查找路径上的任一 x-节点落在同一条垂线上；也不得落在其查找路径上的任一 y-节点所对应的线段上。后一假定条件又将如何消除呢？实际上，正如接下来就要说明的，前面所采用的方法同样可以用来解决这一问题。

算法所构造的查找结构，对应于经变换后的梯形图 Ʈ(φS)。既然如此，进行查询时，也应使用经变换后的待查询点φq。换而言之，整个查找过程中进行的每一次比较，都是在变换后的空间内进行的。至于在变换后的空间内如何进行测试，我们已经很清楚了，可按如下步骤：

在每个 x-节点处，必须按照字典序进行测试。其结果是，不同的点绝不可能落在同一条垂线上。（这里有个问题需要考考读者：既然φ是一个双射，为什么居然能够保证这一点？）这并不意味着；在任何一个 x-节点处的比较结果，不是“位于右侧”，就是“位于左侧”。实际上，测试的结果还可能是“恰好落在直线上”。不过，后一情况的发生，只有一种可能——待查询点与存储在该 x-节点处的端点正好重合。果真如此，这一比较的结论，已经给出了查询的答案！

在各 y-节点处，我们必须将经过变换之后的待查询点，与经过变换之后的线段进行比较。如前所述的比较可能会得出三种结论：“位于上方”、“位于下方”或者 “恰好落在线上”。前两种情况不会有任何问题，我们都可以顺利地前进到该 y-节点两个孩子中的某一个。然而若比较的结果是

“落在线上”，则原先未经变换的点也必然落在未经变换的线段上。同样地，此时的这个结果（即“待查询点落在这条线段上”）也正是查询的答案。

这样，我们就将〖定理 6.3〗的适用范围推广到了由不相交线段组成的任意集合。

〖定理 6.5〗

对于由任意 n 条线段构成的集合 S，算法 TRAPEZOIDALMAP 都可在 $O(n\log n)$期望时间内，计算出 S 的梯形图 $T(S)$以及与之对应的查找结构 D。该结构的期望规模为 $O(n)$；对任一待查询点 q，期望查询时间为 $O(\log n)$。

6.4 *尾分析

〖定理 6.5〗指出，任一待查询点q的期望查询时间为$O(\log n)$。不过，这个结果相当弱。实际上，没有理由指望查找结构的最大查询时间能够很短。一种可能的情况是：无论线段集的排列次序如何，一旦按该次序生成了某个查找结构，就总是相应地存在**某个**待查询点，使得在该结构中查找它需要很长的时间。尽管如此，本节仍将证明，完全不必为此担心——虽然最大查询时间的确可能会很长，但这种情况的概率非常低。为此，首先需要证明以下高概率上界（high-probability bound）。

〖引理 6.6〗

任意给定由 n 条互不相交的线段组成的一个集合 S，一个待查询点 q，以及一个参数$\lambda > 0$。则在由算法 TRAPEZOIDALMAP 构造出来的查找结构中对 q 进行查找时，其查找路径上的节点数目超过 $3\lambda\ln(n+1)$ 的概率，不会超过 $1/(n+1)^{\lambda\ln 1.25-1}$。

〖证明〗

定义一组随机变量X_i，$1 \le i \le n$。若在q所对应的查找路径上，至少有一个节点诞生于算法的第i轮迭代，则X_i为 1；反之，若第i轮迭代没有在该查找路径上增加任何节点，则X_i为 0。不幸的是，如此定义出来的随机变量，并非相互独立的。（在〖定理 6.3〗的证明中，这种独立性并不重要，而这里却必须保证这种独立性。）为此，需要使用一个小技巧。

定义一个有向无环图G，它只有一个源（source），也只有一个渊（sink）。如图 6-15 所示，在G中从源通往渊的每一条路径，都对应于S的一个排列。该图的定义具体如下。对于S的每一子集，G中都有一个对应的节点（其中也包括与空集相对应的一个节点）。我们将略微不那么严格地使用术语——在提到“S的某个子集”时，意思也可能是指“与S的这个子集相对应的那个节点”。一种便于理解的方法就是，假想着将所有节点划分成n+1 层，其中第i层上所有子

集的基数均为i，0 ≤ i ≤ n。请注意，第 0 和n层都分别只有一个节点，分别对应于空集和S本身。第i层上的每个节点，都会发出若干条流出弧（outgoing arc），分别指向位于第i+1 层的各个节点。更准确地说，基数为i的一个子集S'向基数为i+1 的另一个子集S"发出一条弧，当且仅当S' ⊂ S"。也就是说，子集S'向S"发出一条弧，当且仅当只要将S中的某条线段添加到S'中，就可以得到S"。相应地，我们还要用这条（增加的）线段为这条弧做上标记。请注意，位于第i层的任一子集S'，都有且仅有i条流入弧（incoming arc），其中每一条弧都标有S'中的某条线段；另外，S'有且仅有n-i条流出弧，其中每一条弧都标有S\S'中的某条线段。

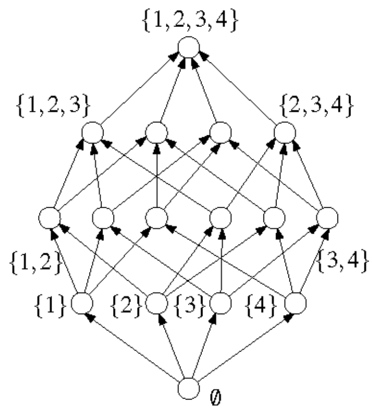


图6-15 有向无环图$\mathcal{G}$

这样，$\mathcal{G}$中从源到渊的每一条有向路径，都分别对应于S的一个排列，也就是算法TRAPEZOIDALMAP可能的一次运行过程。在$\mathcal{G}$中，考察由第i层上的某一子集S'指向第i+1 层上的另一子集S"的一条弧。设该弧的标记为线段s。这样的一条弧的含义是：在S'所对应的梯形图中插入线段s。若在这次插入之后，原先包含q点的那个梯形发生了变化，我们就给这条弧做上记号。按照这种规则，将有多少条弧会被做上记号呢？为了回答这一问题，需要仿照〖定理 6.3〗的证明方法，再次进行后向分析。我们反过来考察这一过程：在将某条弧（所对应的线段）从子集S"中删除之后，点q所在的梯形有可能会发生变化；但是，能够引起这种变化的弧，最多不会超过 4 条。这就意味着，在$\mathcal{G}$中每个节点的所有流入弧中，至多只有四条能够被做上记号。反过来，在某些节点处，这种弧也的确会少于 4 条。若在某个节点处，做有标记的流入弧不足 4 条，我们就可以随便选出若干条（未做标记的）流入弧，强行给它们做上标记——最后必须保证，每个节点都有且仅有 4 条做有标记的流入弧。另外，对于前三层的每个节点来说，流入弧无论如何也不会达到 4 条，这时，我们要给所有的弧都做上标记。

（在算法的运行过程中，）q 所在的梯形不断变化，那么，其变化次数的期望值是多少？我们想要对其做出估计。换而言之，我们要在 $\mathcal{G}$ 中估计出，沿着一条从源通往渊的路径，标有记号的弧的期望数目是多少。为此，定义随机变量 X_i 如下：

$$X_i := \begin{cases} 1 & \text{（若沿着由源通往渊的路径，第}i\text{条弧做有记号）} \\ 0 & \text{（否则）} \end{cases}$$

请注意：这里所定义的X_i，与在 168 页定义的X_i之间存在相似性。沿着这条路径，第i条弧必然起自于某个位于第i-1 层的节点，终止于某个位于第i层的节点——反过来，所有这些（联接于第i-1 层与第i层之间的）弧，成为这条路径上第i条弧的概率都是相等的。位于第i层的每个节点，都有i条流入弧，其中有且仅有 4 条是做有标记的（假定$i \geq 4$）。因此，对于任何$i \geq 4$，都有$\Pr[X_i=1] = 4/i$；若$i < 4$，则有$\Pr[X_i=1] = 1 < 4/i$。此外，我们还可以注意到，X_i是相互独立的（这一点不同于在 168 页定义的随机变量X_i）。

令 $Y := \sum_{i=1}^{n} X_i$。在对应于 q 的查找路径上，节点的总数不会超过 3Y。下面来界定"Y 大于$\lambda \ln(n+1)$"的概率。在此，要利用 Markov 不等式（Markov's inequality）。这一不等式指出：对于任意非负的随机变量 Z 以及任意的$\alpha > 0$，都有

$$\Pr[Z \geq \alpha] \quad \leq \quad \frac{E[Z]}{\alpha}$$

所以，对于任何 $t > 0$，都有

$$\Pr[Y \geq \lambda \mathbf{ln}(n+1)] \quad = \quad \Pr[e^{tY} \geq e^{t\lambda \mathbf{ln}(n+1)}] \quad \leq \quad e^{-t\lambda \mathbf{ln}(n+1)}\, E[e^{tY}]$$

读者应该还记得：若干随机变量之和的期望值，等于这些随机变量各自的期望值之和。乘积的期望值，通常并不等于期望值的乘积。但只要各随机变量相互独立，则乘积的期望值的确等于期望值的乘积。在这里，随机变量 X_i 的确是相互独立的，故有：

$$E[e^{tY}] \quad = \quad E[e^{t\sum_{i=1}^{n} X_i}] \quad = \quad E[\prod_{i=1}^{n} e^{tX_i}] \quad = \quad \prod_{i=1}^{n} E[e^{tX_i}]$$

若取 $t = \mathbf{ln}1.25$，则得

$$E[e^{tX_i}] \quad \leq \quad e^t \cdot \frac{4}{i} + e^0 \cdot (1-\frac{4}{i}) \quad = \quad (1+\frac{1}{4}) \cdot \frac{4}{i} + 1 - \frac{4}{i} \quad = \quad 1 + \frac{1}{i} \quad = \quad \frac{1+i}{i}$$

进而

$$\prod_{i=1}^{n} E[e^{tX_i}] \quad \leq \quad \frac{2}{1} \times \frac{3}{2} \times \ldots \times \frac{n+1}{n} \quad = \quad n + 1$$

综上所述，即可得到我们所希望证明的上界：

$$\Pr[Y \geq \lambda \mathbf{ln}(n+1)] \quad \leq \quad e^{-\lambda t \mathbf{ln}(n+1)} \cdot (n+1) \quad = \quad \frac{n+1}{(n+1)^{\lambda t}} \quad = \quad 1 / (n+1)^{\lambda t - 1}$$

❑

借助这个引理，就可以得到最大期望查询时间的一个上界。

〖引理 6.7〗

任意给定由 n 条互不相交的线段组成的一个集合 S，以及一个参数$\lambda > 0$。则在算法 TRAPEZOIDALMAP 为 S 构造出来的查找结构中，查找路径的最大长度超过 $3\lambda \mathbf{ln}(n+1)$的概率不会超过 $2 / (n+1)^{\lambda \mathbf{ln}1.25-3}$。

〖证明〗

若两个待查询点 q 和 q'在查找结构 $\mathcal{D}$ 中对应的查找路径相同，则称之为等价的（equivalent）。S 中的每个端点都分别发出一条垂线，从而将整个平面分割为若干垂直的条带（slab）。如果再考虑条带与 S 中各线段所有可能的交，则每根条带都将进一步被划分为多个梯形。对平面的这种分解，最多会生成 $2(n+1)^2$ 个梯形。无论为 S 构造出来的具体是哪一种查找结构，位于同一梯形内部的任意两点都必然是等价的。这是因为，在查找过程中所进行的比较不外乎两种：相对于经过某个端点的一条垂线，通过比较判断出某个待查询点是位于其左侧，还是右侧；或者，相对于某条线段，通过比较判断出某个待查询点是位于其上方，还是下方。

这就意味着：如果要界定$\mathcal{D}$中查找路径的最大长度，只要逐一地考察最多 $2(n+1)^2$ 个待查询点就足够了——其中各点，分别来自这 $2(n+1)^2$ 个梯形之一。根据〖引理 6.6〗，对于固定的某个待查询点而言，其查找路径的长度超过 $3\lambda \mathbf{ln}(n+1)$的概率，不会超过 $1 / (n+1)^{\lambda \mathbf{ln}1.25-1}$。在最坏情况下，这 $2(n+1)^2$ 个测试点中至少存在一个点，其对应的查找路径长度超出这一范围的概率不会超过 $2(n+1)^2 / (n+1)^{\lambda \mathbf{ln}1.25-1}$。 ❑

这一引理说明，最大期望查询时间为 $O(\mathbf{log}n)$。比如，若取$\lambda = 20$，则 $\mathcal{D}$ 中各条查找路径的最大长度超过 $3\lambda \ln(n+1)$的概率，将不会超过 $2 / (n+1)^{1.4}$。在 $n > 4$ 后，这一数值将小于 1 / 4。也就是说，$\mathcal{D}$ 的查询时间性能很好的概率至少有 3 / 4。按照类似的方法也可以证明，$\mathcal{D}$ 的规模不超过 $O(n)$的概率也至少有 3 / 4。我们已经注意到，若查找结构中各查找路径的最大长度是 $O(\mathbf{log}n)$，而且查找结构的规模为 $O(n)$，则算法的运行时间必然不会超过 $O(n\mathbf{log}n)$。因此，查询时间、查找结构的规模及其构造时间性能都很好的概率，至少有 1/2。

现在，我们也可以构造出另一个查找结构，其最坏情况下的查询时间为 $O(\mathbf{log}n)$，而且在最坏情况下占用的空间规模为 $O(n)$。构造方法如下。首先对集合 S 运行算法 TRAPEZOIDALMAP，在构造的过程中，跟踪记录查找结构所占用空间的大小、查找路径的最大长度。在适当地选取好两个常数 c_1 和 c_2 之后，一旦发现查找结构的规模超过了 $c_1 n$，或者路径深度超过了 $c_2 \mathbf{log}n$，就立即停止算法，然后重新运行一次——当然，需要重新对各线段做随机排列。按照某种排列次序所生成的查找结构，其占用空间的规模、（最大）查找深度均满足要求的概率不会低于 1 / 4。因此，我们可以期望在 4 次尝试之后，就得出一个满意的结果。（实际上，对于足够大的 n，这一概率几乎为 1，因此，所需尝试的次数只比 1 略多一点。）由此可以归纳得到如下结论：

〖定理 6.8〗

任意给定由 n 条边组成的一个平面子区域划分。必然存在支持对 S 进行点定位查询的一个数据结构，它只占用 O(n)的存储空间，而且在最坏情况下查找时间不超过 O(logn)。

按照这里所取的常数，每次查找需要访问的节点数多达 60**ln**n，这一结果不仅难以令人十分信服，而且也不会让人感兴趣。不过，只要利用同样的技巧，完全可以得出更好的常数。

关于预处理所需的时间，上述定理并没有给出任何说明。的确，为了能够跟踪记录查找路径的最大长度，我们需要同时兼顾多达 $2(n+1)^2$ 个测试点——参见〖引理 6.7〗。然而事实上，这个数字可以减少到O(n**log**n)，从而将预处理所需的时间降至$O(n\log^2 n)$。

6.5 注释及评论

在计算几何（computational geometry）领域，点定位这一问题的历史相当悠久。Preparata和Shamos[323]曾对这方面的早期研究成果做过综述。这个问题的解决方法多种多样，其中有四种各自不同的方法，性能都可以达到O(**log**n)查询时间、O(n)存储空间的水平。这些方法是：Edelsbrunner等人 [161]基于线段树（segment tree）和分散层叠（fractional cascading，参见第 10 章）等技术提出的链方法（chain method）；Kirkpatrick[236]提出的三角形细分法（triangle refinement method）；Sarnak和Tarjan[336]以及Cole[135]基于持续性（persistency）的方法；还有Mulmuley[289]所提出的随机增量式算法。这里沿用了Seidel[345]的讲解过程以及对Mulmuley算法的分析方法。

近来这方面的研究工作，有很多都转向了动态点定位（dynamic point location）的问题[41][115][120]——在这类问题中，允许通过增加或删除边，对子区域划分本身进行（动态的）修改。针对动态点定位的问题，Chiang和Tamassia[121]曾经做过综述。

高于二维的点定位问题，依然没有彻底解决。Preparata和Tamassia[324]曾设计一种通用的结构，可支持三维空间中的凸子区域划分。也可将动态平面点定位结构（dynamic planar point-location structure）与持续性（persistency）技术相结合，得到一种占用O(n**log**n)空间的静态三维点定位结构（static three-dimensional point location structure），其查询时间为$O(\log^2 n)$[361]。只需线性的空间、查询时间为O(**log**n)的结构尚未发现。在更高维的空间中，只针对某些特殊的子区域划分找到了高效的点定位结构，比如由超平面构成的排列（arrangement）[95][104][131]。在d维空间中，设H为由n张超平面构成的集合，关于由H导出的那个子区域划分，一个众所周知的结果就是：在最坏情况下，该子区域划分的组合复杂度（combinatorial complexity，即其中顶点、边等元素的总数）为$\Theta(n^d)$[158]——也可进一步参考第 8 章的“注释及评论”一节。Chazelle和Friedman[104]已证明：只需$O(n^d)$空间即可

存储这种子区域划分；而且，可以在$O(\log n)$时间内完成每次点定位查询。已找到高效点定位结构的其它一些特殊的子区域划分，包括凸多胞体（convex polytope）[131][266]、三角形的排列（arrangement of triangle）[59]以及代数簇的排列（arrangement of algebraic varieties）[102]。

在三维或者更高维的空间中，有一些点定位问题也能够得到有效地求解，在这类问题中，有些需要对（子区域划分中）各单元的形状做某种（强制性）假设。其中的两个例子就是矩形子区域划分（rectangular subdivision）[57][162]和所谓的丰满子区域划分（fat subdivision）[302][309]。

点定位查询所要求的输出，通常都是在子区域划分中包含某个给定待查询点的那个单元的编号。比如在d维空间中，所谓针对某个凸多胞体的点定位，可能的查找结果不外乎两种：待查询点落在多胞体的内部，或者外面。因此，可以指望找到某种支持点定位操作的数据结构，其占用的存储量远远低于这种子区域划分本身的组合复杂度。比如由n个半空间的交所确定的凸多胞体，只需要$\Theta(n^{\lfloor d/2 \rfloor})$空间 [158]——可进一步参考第 11 章的“注释及评论”一节。事实上，的确存在某种只占用$O(n)$空间的数据结构，借助它可在$O(n^{1-1/\lfloor d/2 \rfloor}\log^{o(1)} n)$时间内完成每次点定位查询 [264]。而对于平面上线段的排列，还有其它的几种数据结构能够支持隐式点定位（implicit point location）查询。虽然线段排列的组合复杂度高达平方量级，但是这几种数据结构却只需低于平方量级的空间 [160][7]。

6.6 习题

习题 6.1 将如 图 6-16 所示的一组线段按照某种次序逐一插入，以构造出与该集合对应的查找结构$\mathcal{D}$。试画出该查找结构。

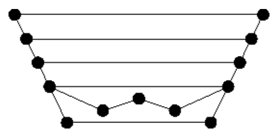


图6-16 一组线段

习题 6.2 试给出 n 条线段以及它们的一个次序。按照这一次序，由（本章的）算法构造出来的查找结构将占用$\Theta(n^2)$空间，而且最坏情况下的查询时间为$\Theta(n)$。

习题 6.3 本章讨论了如何通过预处理来解决点定位问题。然而，倘若是一次性计算问题（single shot problem）——亦即，子区域划分和待查询点是同时给定的——则无法借助某种预处理来加速查找。本题以及接下来的几题，将对这类问题做一讨论。

任意给定由 n 个顶点组成的一个简单多边形（simple polygon）$\mathcal{P}$，以及一个待查询点 q，可以按照下面的算法，判断出 q 是否落在 $\mathcal{P}$ 的内部。考虑一条射线$\rho := \{(q_x + \lambda, q_y) : \lambda > 0\}$——也就是从 q 出发、水平向右的那条射线。对于 $\mathcal{P}$ 的每一条边 e，判断 e 是否与ρ相交。若与ρ相交的边共有奇数条，则 $q \in \mathcal{P}$；否则，$q \notin \mathcal{P}$。

试证明，上述算法是正确的。另外，请说明应该如何处理各种退化情况。（其中的一种退化情况就是，ρ经过某条边的一个端点。是否还有其它的特殊情况呢？）这个算法的运行时间是多少？

习题 6.4 试证明：任意给定一个包含 n 个顶点的平面子区域划分 S，以及一个待查询点 q，我们都可以在 $O(n)$时间内，从 S 中找出 q 所在的那张面。这里，假定 S 已经被表示为双向链接边表的形式。

习题 6.5 给定一个凸多边形（convex polygon）$\mathcal{P}$，它的 n 个顶点已经按照沿 $\mathcal{P}$ 边界的次序，存储为一个有序数组。试证明：对于任一给定的待查询点 q，都可以在 $O(\log n)$时间内，判断出 q 是否落在 $\mathcal{P}$ 的内部。

习题 6.6 给定一个 y-单调多边形 $\mathcal{P}$，它的 n 个顶点已经按照沿 $\mathcal{P}$ 边界的次序，存储为一个有序数组。你能否将上一题的结果，推广到 y-单调多边形？

习题 6.7 如 图 6-17 所示，一个多边形$\mathcal{P}$是所谓星形多边形（star-shaped polygon）的条件是：在$\mathcal{P}$中存在一个点p，使得对于$\mathcal{P}$中其它的任何点q，线段$\overline{pq}$完全落在$\mathcal{P}$中 [①]。假如给定一个星形多边形$\mathcal{P}$，以及其中满足上述条件的一个点p。另外，与前面的两道题一样，$\mathcal{P}$的n个顶点已经按照沿$\mathcal{P}$边界的次序，存储为一个有序数组。

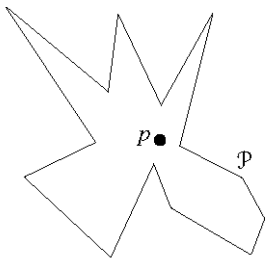


图6-17 星形多边形

试证明：任给一个待查询点 q，都可在 $O(\log n)$时间内，判断出 q 是否落在 $\mathcal{P}$ 的内部。若只告诉我们 $\mathcal{P}$ 是一个星形多边形，而没有进一步给出点 p，情况又将如何？

习题 6.8 试设计一个确定性算法，计算任意一组互不相交的线段所对应的梯形图。确定性算法（deterministic algorithm）的要求是，在算法过程中不以随机策略来进行选择。可以采用第 2 章所介绍的平面扫描算法模式。其最坏情况复杂度必须是$O(n\log n)$。

习题 6.9* 试给出一个随机算法（randomized algorithm），在 $O(n\log n + A)$的期望时间内，在一组共 n 条线段内，找出所有两两相交的线段。其中，A 为相交线段对的数目。

习题 6.10 试设计一个算法，在 $O(n\log n)$时间内解答如下问题：给定由 n 个点构成的一个集合 P，

[①] 对于每个星形多边形 $\mathcal{P}$，具有这一性质的所有点构成的集合也被称作 $\mathcal{P}$ 的核集（kernel），简称核。——译者

找出一个数$\varepsilon > 0$，使得在经过剪切变换$\varphi : (x, y) \mapsto (x+\varepsilon y, y)$之后，原先 x-坐标不同的各点，在 x-方向上的次序不变。

习题 6.11 设 S 为由平面上若干互不相交的线段组成的一个集合，设 s 为与 S 中各线段均不相交的一条新线段。试证明：$\mathcal{T}(S)$中的一个梯形Δ在$\mathcal{T}(S\cup\{s\})$中依然保持不变，当且仅当 s 与Δ的内部不相交。

习题 6.12 试证明：在算法 TRAPEZOIDALMAP 的第 i 轮迭代中，查找结构 $\mathcal{D}$ 所含内部节点的数目将增加 $k_i - 1$。这里，k_i为$\mathcal{T}(S_i)$中新生出的梯形数目（亦即 $\mathcal{D}$ 中新生出的叶子数目）。

习题 6.13 按照平面扫描的论证方法，试证明：在处于一般性位置的 n 条线段所对应的梯形图中，至多含有 3n + 1 个梯形。（可以假想着有一条垂线自左向右扫过整个平面，在各线段的每一端点处，都要停留片刻。统计出该扫描线所经过的梯形数目。）

习题 6.14 对于由 n 条线段组成的集合 S，我们所定义的梯形图要求 S 必须处于一般性位置。试针对任意的线段集合，给出梯形图的一个（一般性的）定义。试证明：其中梯形数目的上界仍然为 3n + 1。

习题 6.15 我们最初遇到的点定位问题，是针对地球表面而言的。尽管如此，我们所解决的却只是平面的点定位问题。然而，地球毕竟是圆的。既然如此，又该如何对球面子区域划分（spherical subdivision，也就是说，对球面进行的子区域划分）进行定义呢？试针对这类子区域划分，给出一种支持点定位操作的数据结构。

习题 6.16 光线发射问题（ray shooting problem）来自于计算机图形学领域（参见第 8 章）。在二维情况下，这个问题可以描述如下：对于由任意n条互不相交的线段组成的集合S，通过适当的预处理，使得如下查询可以很快得到回答：“给定一条待查询光线ρ（也就是从某一点发出的一条射线），在S中找到ρ所穿过的第一条线段。”（应该如何处理这里的退化情况？这个问题也留给你。）

在本题中，我们只考虑所谓的垂直光线发射问题（vertical ray shooting problem）——具体而言，也就是要求待查询的光线必须是垂直向上发出的。这样，只要给出（射线的）起始位置，就可以定义这样的一次查找。

试给出一种数据结构，针对由任意 n 条处于一般性位置、互不相交的线段组成的集合 S，解答相应的垂直光线发射问题。另外，你所给出的这一数据结构在查询时间、存储空间方面的上界各是多少？试给出证明。所需的预处理时间呢？

习题 6.17* 〖定理 6.8〗还有另一个更为精确的版本，它不是笼统地以数量级的概念（大O记号）来确定节点数目以及查找结构深度的上界。试给出并证明该定理的这一版本。为了得出一个较之〖定理 6.8〗更为准确的常数，你需要对本章有关〖定理 6.8〗的证明中的某些细节做些调整。

7

Voronoi图：邮局问题

如果你是某个咨询委员会的成员，正在为一个超级市场连锁系统做规划，需在某处开设一家新的分店。为了评判该分店是否有利可图，你必须估计出它将能够吸引多少顾客。为此，你必须建立一个模型，以描述潜在顾客的行为：人们将如何选择采购地点？在依据社会地理学（social geography）研究某一国家内部的经济行为时，也会遇到一个类似的问题：应该如何界定不同城市各自对应的商业范围？在做了进一步抽象处理之后，我们将得到一组中心位置，这些被称作**基点**（site）的位置，可以提供某种商品或者服务。而我们所希望了解的是，愿意从某一基点得到商品或者服务的人们都

居住在什么范围之内（按计算几何的习惯，这些基点都比作邮局，顾客们到邮局去是为了投寄自己的邮件——这也是本章副标题的由来）。为讨论这一问题，需做如下简化假设：

- 即使是在不同的基点，同一商品或服务的价格也是相同的；
- 为获得任何一种商品或者服务，需要支付的费用等于其本身的价格，再加上为抵达该基点需要支付的交通费用；
- 到达某一基点所需的费用，等于（顾客）与该基点之间的欧氏距离乘以一个固定的单位距离价格；
- 在寻求某种商品或者服务时，顾客们都会使其代价最小化。

通常，上述假设并不能完全满足：同一种商品，在某个基点的价格可能会高于另一个基点；往来于两地之间所需要的费用，可能并不正比于其间的欧氏距离。不过做为近似，上述模型还是能够粗略地表示不同基点所对应的商业范围。如果在某些区域内，人们的行为与根据该模型得出的预测有所出入，那么就可以通过进一步的研究，以确定是什么原因导致了人们行为上的这种差异。

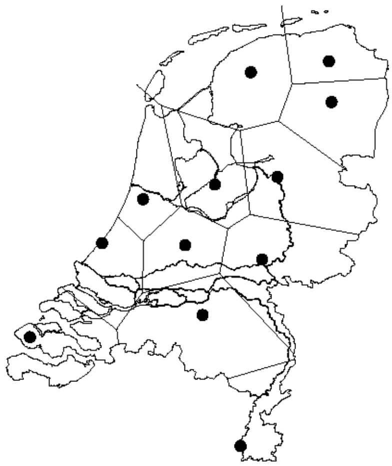


图7-1 根据Voronoi分配模型，为荷兰12个省的首府分别确定的商业范围

我们所感兴趣的，是上述模型的几何解释。根据该模型的假设条件，可以得出对整个被考察区域的一种子区域划分（subdivision）。这种划分的结果是，该区域被剖分为多个子区域（region）——即与各基点分别对应的商业范围——而生活在同一子区域之内的所有人，都会前往同一基点采购。根据我们的假设，人们都会无一例外地前往最近的基点购物——这是一个相当理想的情况。这意味着：在对应于某个基点的商业范围之内，任一位置距离该基点都要比距离其它基点更近。图 7-1 给出了这样的一个例子。该图中的基点，分别是荷兰 12 个省的首都。

这种模型将每个点都分配给与之距离最近的那个基点，我们称这一原则为 Voronoi 分配模型（Voronoi assignment model）。根据这一模型得出的子区域划分，被称为这一基点集合的 Voronoi 图（Voronoi diagram）。有了 Voronoi 图，也就获得了这些基点各自对应的商业范围，以及它们之间关

系的所有信息。例如，要是有两个子区域相邻于一段共同的边界，那么其对应的两个基点之间就很可能存在着直接的竞争，它们要争夺生活在这段边界附近的那些客户。

Voronoi图是一种通用的几何结构。我们只介绍了其在社会地理学中的一个应用，实际上在物理学、天文学、机器人学（robotics）以及众多的领域，Voronoi图都有着广泛的应用。此外，它还与另一种重要的几何结构——所谓的Delaunay三角剖分（Delaunay triangulation）——有着密切的联系，第 9 章将讨论后一种结构。本章的讨论范围只限制于平面点集的Voronoi图，并研究其基本性质及构造算法。

7.1 定义及基本性质

任意两点 p 和 q 之间的欧氏距离，记作 dist(p, q)。就平面情况而言，我们有

$$dist(p, q) = \sqrt{(p_x-q_x)^2 + (p_y-q_y)^2}$$

设 $P := \{p_1, \ldots, p_n\}$为平面上任意 n 个互异的点；这些点也就是基点。按照我们的定义，所谓 P 对应的 Voronoi 图，就是平面的一个子区域划分——整个平面因此被划分为 n 个单元（cell），它们具有这样的性质：任一点 q 位于点 p_i 所对应的单元中，当且仅当对于任何的 $p_j \in P$，$j \neq i$，都有 $dist(q, p_i) < dist(q, p_j)$。我们将与 P 对应的 Voronoi 图记作 Vor(P)。我们对这一术语的使用，可能会有些不甚严格——有的时候，“Vor(P)”或者“Voronoi 图”所指示的仅仅只是组成该子区域划分的边和顶点。例如，当我们说“Voronoi 图是连通的”时，意思是指“这些边和顶点构成了一个连通集”。在 Vor(P)中，与基点 p_i 相对应的单元记作 $\mathcal{V}(p_i)$——称作与 p_i 相对应的 Voronoi 单元（Voronoi cell）。（按照本章开头导言部分的术语，$\mathcal{V}(p_i)$也就是与基点 p_i 相对应的商业范围。）

下面将对 Voronoi 图做进一步考察。首先，要对单个 Voronoi 单元的结构做一研究。任给平面上两点 p 和 q，所谓 p 和 q 的平分线（bisector），就是线段 $\overline{pq}$ 的垂直平分线。该平分线将平面划分为两张半平面（half-plane）。点 p 所在的那张开半平面记作 h(p, q)，点 q 所在的那张开半平面记作 h(q, p)。请注意，$r \in h(p, q)$当且仅当 $dist(r, p) < dist(r, q)$。据此，可以得出如下观察结论：

〖观察结论 7.1〗

$$\mathcal{V}(p_i) = \bigcap_{1\leq j\leq n,\, j\neq i} h(p_i, p_j)$$

也就是说，$\mathcal{V}(p_i)$是(n-1)张半平面的公共交集；进而，如 图 7-2 所示，它也是一个（不见得有界的）开的凸多边形（convex polygon）子区域，而且沿着其边界至多有n-1 个顶点和n-1 条边。

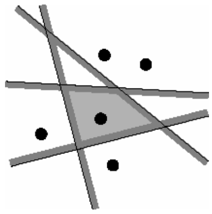


图7-2 (n-1)张半平面的公共交集

Voronoi图的全貌又是什么样子呢？我们已经看到，该图中的每一单元都是若干张半平面的交，因此，如 图 7-3 所示，在Voronoi图这种平面子区域划分中，所有的边都是直的。其中，有的是线段，有的则是射线[①]。除非所有的基点都共线，否则任何边都不会是一条完整的直线。

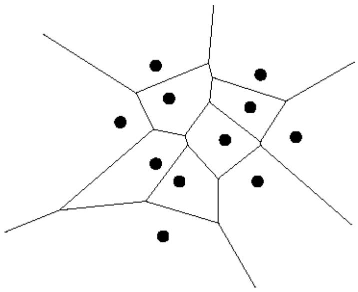


图7-3 Voronoi图

〖定理 7.2〗

给定由平面上任意 n 个点构成的集合 P。若所有点都共线，则 Vor(P)由 n-1 条平行直线构成；否则，Vor(P)将是连通的，而且其中的边不是线段就是射线。

〖证明〗

定理的前一种情形很容易得证，故这里假定并非所有的点都共线。

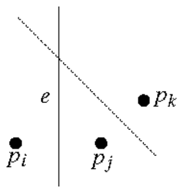


图7-4 除非所有基点都共线，否则Vor(P)中不包含完整的直线

首先来说明：Vor(P)中的边都是线段或者射线。我们已经知道，Vor(P)的每条边都属于某条直线——具体而言，也就是位于每一对基点之间的某条平分线——的一部分。如 图 7-4 所示，假设与定理的断言相反，Vor(P)中包含一条完整的直线e。设来自于Voronoi单元$V(p_i)$和$V(p_j)$的（共

[①] half-line，即射线。——译者

同）边界。任取不与p_i和p_j共线的第三个点$p_k \in P$。p_j与p_k之间的平分线不可能与e平行，换而言之，它必定与e相交。然而，e落在$h(p_k, p_j)$内部的那一段又不可能属于$V(p_j)$的边界——因为，这一段上的任何点到p_k的距离，都要比到p_j的距离更短。这与假设不合。

下面只需证明，Vor(P)是连通的。假设不是如此，则存在某个 Voronoi 单元 $V(p_i)$，将整个平面分割成两个（互不相交的）部分 。鉴于任何 Voronoi 单元都是凸集，$V(p_i)$只可能是一个（两端无穷的窄）条形区域，其边界是两条完整的平行直线。然而以上刚刚证明过，Voronoi 图中的任何边都不可能是一条完整的直线，矛盾。 ❑

在了解了 Voronoi 图的结构之后，现在来对其复杂度（即其中包含的顶点与边的数目）做一分析。既然共有 n 个基点，而且每个 Voronoi 单元最多有 n-1 个顶点和 n-1 条边，故 Vor(P)的复杂度不会超过平方量级。

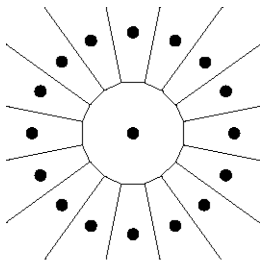


图7-5 Vor(P)中单个单元的复杂度可能高达$O(n)$

然而，究竟Vor(P)的复杂度能否达到平方量级呢？答案并非一目了然。我们很容易就可以构造出一个实例，其中的某一个Voronoi单元的确具有线性的复杂度（如 图 7-5 所示）。然而，这种具有线性复杂度的单元是否会同时出现很多个 [①]呢？下面这则定理回答说：这种情况是不可能的；而且平均算来，每个Voronoi单元的顶点数要小于 6。

〖定理 7.3〗

若 $n \geq 3$，则在与平面上任意 n 个基点相对应的 Voronoi 图中，顶点的数目不会超过 2n-5，而且边的数目不会超过 3n-6。

〖证明〗

若所有基点都共线，则根据〖定理 7.2〗可直接得出该定理的结论。现在假定不是这种情况。此时为了证明这一定理，需要借助欧拉公式，该公式指出：对任何连通的平面嵌入图（planar embedded graph），其顶点数m_d、边数m_e以及面数m_f必然满足如下关系：

[①] 也就是线性个。比如 n/c 个，其中的 c 为常数。——译者

$m_d - m_e + m_f = 2$[①]

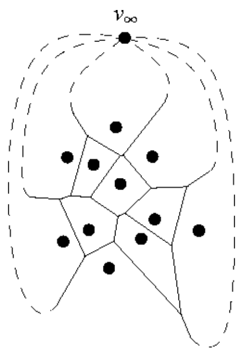


图7-6 在“无穷远处”引入附加顶点v_∞

欧拉公式还不能直接应用于Vor(P)，因为在Vor(P)中存在单向无穷边（half-infinite edge），故它还不是真正的图。为将条件补齐，除原有的顶点之外，可在“无穷远处”引入一个附加的顶点v_∞（如 图 7-6 所示）；然后，将Vor(P)中的所有单向无穷边与这个点相联。这样就得到了一幅连通的平面图，而欧拉公式可以适用于这幅图。于是，我们就得到了Vor(P)的顶点数n_v、边数n_e以及基点数n之间的关系：

$$(n_v + 1) - n_e + n = 2 \quad \text{......(7.1)}$$

此外，在这幅经过拓展的图中，每条边都恰好对应于两个顶点，因此若将所有顶点的度数累加起来，得到的就是两倍的边数。因为包括 v_∞在内的每个顶点的度数不低于 3，故有

$$2n_e \geq 3(n_v + 1) \quad \text{......(7.2)}$$

这一不等式与等式(7.1)合起来，就得到了本定理。 ❑

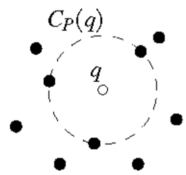


图7-7 q关于P的最大空圆

本节最后，我们对Voronoi图中各边及顶点的特性做一刻画。我们知道，其中每条边都是某对基点之间平分线的一段，而其中每一顶点都是某两条平分线的交点。虽然这种平分线多达平方量级条，但是Vor(P)的复杂度却只是线性的。因此，并非所有的平分线都能为Vor(P)贡献一条边，而且它们之间的交点也并不都是Vor(P)的顶点。为挑选出确定Vor(P)形状的那些平分线和交点，如 图 7-7 所示，

① 一般地，若该图由 m_c个连通块组成，则有 $m_d - m_e + m_f - m_c = 1$。——译者

对于任何点q，我们将以q为中心、内部[①]不含P中任何基点的最大圆，称作**q关于P的最大空圆**（largest empty circle），记作$C_P(q)$。以下定理指出了Voronoi图的顶点及边所具有的特征：

〖定理 7.4〗

对于任一点集 P 所对应的 Voronoi 图 Vor(P)，下列命题成立：

1) 点 q 是 Vor(P)的一个顶点，当且仅当在其最大空圆 $C_P(q)$的边界上，至少有三个基点；

2) p_i 和 p_j 之间的平分线确定了 Vor(P)的一条边，当且仅当在这条线上存在一个点 q，$C_P(q)$的边界经过 p_i 和 p_j，但不经过其它基点。

〖证明〗

(i) 假如某个点q，$C_P(q)$的边界经过至少三个基点。从中选出三个基点p_i、p_j和p_k。既然$C_P(q)$的内部是空的，故q必然同时落在$\mathcal{V}(p_i)$、$\mathcal{V}(p_j)$和$\mathcal{V}(p_k)$的边界上，因此q肯定是Vor(P)的一个顶点（如 图 7-8 所示）。

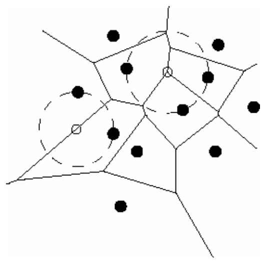


图7-8 最大空圆$C_P(q)$必然由三个基点确定

反之，Vor(P)中的每个顶点都与至少三条边相关联，因此也至少与三个Voronoi单元相关联，不妨设这三个单元分别为$\mathcal{V}(p_i)$、$\mathcal{V}(p_j)$和$\mathcal{V}(p_k)$。顶点q到p_i、p_j和p_k的距离相等，而且到其它各基点的距离都不可能更近——否则，$\mathcal{V}(p_i)$、$\mathcal{V}(p_j)$和$\mathcal{V}(p_k)$就不可能相交于q[②]。于是，边界由p_i、p_j和p_k确定的这个圆，内部不可能包含任何基点。

(ii) 假如某个点 q 具有本定理所述的性质。既然 $C_P(q)$的内部不包含任何基点，且 p_i 和 p_j 落在其边界上，故对任何 $1 \le k \le n$，都应有 $dist(q, p_i) = dist(q, p_j) \le dist(q, p_k)$。于是，q 必然落在 Vor(P)某条边或者某个顶点上。而根据定理的前半部分，q 不可能是 Vor(P)的一个顶点。因此，q 只能落在 Vor(P)某条边上，且该边是 p_i 和 p_j 之间平分线上的一段。

反过来，假设 p_i 和 p_j 之间的平分线确定了一条 Voronoi 边。来自这条边内部的任何一个点 q，其对应的最大空圆的边界必然经过 p_i 和 p_j，但不经过其它基点。 ❑

[①] 也就是说，除去该圆的边界。——译者

[②] 此处不甚准确。根据前面的定义，任何 Voronoi 单元都是开集，故任何两个单元都不相交。更严格的表述应该是：这三个 Voronoi 单元的闭包不可能相交于 q。——译者

7.2 构造 Voronoi 图

前一节讨论了Voronoi图的结构，下面着手来构造它。由〖观察结论 7.1〗可得出一个简明的方法：采用第 4 章的算法，为每一基点p_i构造出所有半平面$h(p_i, p_j)$的交，$j \neq i$。如此，将为每一Voronoi单元花费$O(n\log n)$时间，故构造整个Voronoi图共需$O(n^2\log n)$时间。可否更快？毕竟，Voronoi图本身只不过是线性复杂度。答案是肯定的：以下介绍的平面扫描（plane sweep）算法——自诞生起一般都称作**Fortune算法**——可在$O(n\log n)$时间内构造整个Voronoi图。或许你会试图寻找更快（如线性时间）的算法。但实际上这是非分的奢望：n个实数的排序问题，可归约[①]为Voronoi图的构造问题，故在最坏情况下，任何此类算法都至少需要$\Omega(n\log n)$时间。换而言之，Fortune算法已是最优。

平面扫描算法的策略，就是用一条水平直线——称为**扫描线**（sweep line）——自上而下扫过整个平面。在扫描的过程中需要维护某些信息，这些信息与我们希望构造的结构相关。更准确地说，需要维护的是该结构与扫描线相交部分的信息。在扫描线向下移动的过程中，只有在某些特定的位置——称为**事件点**（event point）——这种信息才需要更新。

现在应用这种一般性的策略，来计算与平面点集 $P = \{p_1, p_2, \ldots, p_n\}$相对应的 Voronoi 图。按照平面扫描的算法模式，我们用一条水平直线 l 自上而下扫过平面。这一模式需要维护 Voronoi 图与当前扫描线的相交部分。遗憾的是，这并非一桩易事——因为，Vor(P)位于扫描线 l 上方部分的结构，不仅取决于位于 l 上方的那些基点，而且也取决于 l 下方的某些基点。换而言之，当扫描线到达 Voronoi 单元 $V(p_i)$的最高顶点时，它与该单元对应的基点之间依然有一段距离。因此，此时我们还没有掌握确定该顶点所需要的全部信息。在这种情况下应用平面扫描的算法模式，其形式需要略做变通：我们所维护的信息，将不再是 Voronoi 图与扫描线相交的部分，而是与位于 l 上方的那些基点相对应的 Voronoi 图的局部——无论位于 l 下方的那些基点位置如何，都不会对这些部分有任何影响。

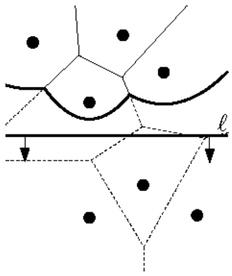


图7-9 扫描线l以及其上方不再会发生变化的部分

位于l上方那个闭的半平面记作l^+。如 图 7-9 所示，在l之上，Voronoi图的哪些部分将不再会发生变化呢？也就是说，对于哪些点$q \in l^+$，已经可以确定与之距离最近的基点呢？点$q \in l^+$到位于l下方

① 更严格的叙述应该是：实数排序问题“可以在线性时间内归约为”（be linear-time reducible to）构造Voronoi图的问题。——译者

的任一基点的距离，都要大于q到l本身的距离。因此，倘若存在某个基点$p_i \in l^+$，使得q到p_i的距离不超过q到l的距离，那么与q相距最近的那个基点，就不可能出现在l下方。距离某个基点$p_i \in l^+$比距离l更近的点所构成的集合，其边界是一条抛物线。于是，距离位于l之上的每个基点都要比距离l更近的那些点所构成的集合，其边界必然由若干段抛物线弧确定（如 图 7-10 所示）。

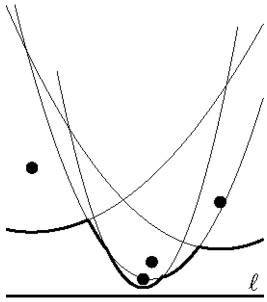


图7-10 海滩线必然由若干段抛物线弧共同围成

这一组依次相联的抛物线弧，称作海滩线（beach line）。可以从另一个角度来想象海滩线的样子：位于扫描线之上的每个基点 p_i，都（与扫描线共同）确定了一条完整的抛物线β_i；而所谓的海滩线，就是这样一个函数：对于任一 x-坐标，该函数的取值都是这些抛物线中的最低者。

〖观察结论 7.5〗

海滩线沿 x 方向单调——即，它与任一垂线相交而且仅相交于一点。

不难发现，有的抛物线可为海滩线贡献多段弧。同一抛物线最多可贡献多少段弧，将在后面考虑。组成海滩线的抛物线弧依次首尾相联，其接合点称为断点（breakpoint）。你可能注意到，每个断点都落在某条 Voronoi 边上。这并非巧合，随着扫描线自上而下扫过整个平面，所有断点的轨迹合起来恰好就是待构造的 Voronoi 图。借助基本的几何知识，即可证明海滩线所具有的这些特性。

既然如此，在扫描线 l 的移动过程中，我们维护的并非 Vor(P)与 l 相交的部分，而是海滩线。当然，无法显式地维护海滩线——因为，随着 1 的运动它会持续不断地更新。那么，应该如何表示海滩线结构呢？其组合结构的变化规律目前尚不清楚，故暂将这个问题搁到一边。所谓海滩线的组合结构发生变化，指的是其上出现了新的抛物线弧，或原有的某段抛物线弧收缩成一个点并进而消失。

首先考虑对应于海滩线上出现新弧的事件，也就是在扫描线l触及某一新基点的时刻。在此刹那，该基点对应于一条宽度为零的退化抛物线——亦即，将该新基点与扫描线联接起来的垂直线段 ①。随着扫描线继续下移，该抛物线将逐渐伸展开来，其位于此前海滩线下方的部分，将成为当前海滩线上的一段。图 7-11 显示了这一过程。对应于新基点的这类事件，称作**基点事件**（site event）。

---

① 准确地讲，新诞生的抛物线应该是一条垂直向上的（无穷）射线。原文所讲的“线段”，对应于这条退化的抛物线对海岸线最初所贡献的一段。——译者

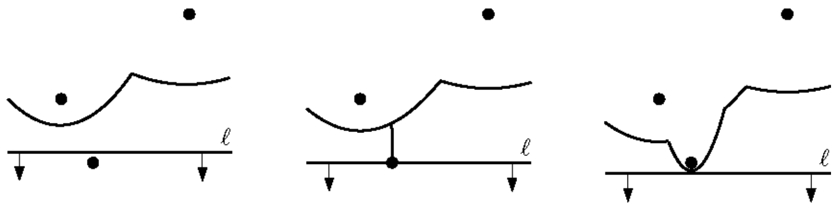


图7-11 当扫描线遇到一个基点时，海滩线上会出现一段新的弧

当基点事件发生时，Voronoi 图会相应地发生什么变化呢？ 我们记得，沿海滩线上各个断点的运动轨迹，就勾勒出了 Voronoi 图的各边。每发生一次基点事件，就会生成两个新的断点，此后它们会逐渐地勾勒出同一条新边。

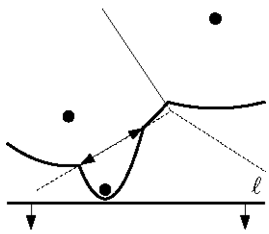


图7-12 一对断点逐渐相互远离，其轨迹勾勒出同一条边

实际上，在刚刚诞生的那一瞬间，这两个断点相互重合，然后才会各自朝相反的方向运动，而且它们所勾勒的都是同一条边（如 图 7-12 所示）。在一开始，这条边与Voronoi图位于扫描线之上的其它部分并不相联。随着这条边的不断生长，直到后来——我们不久就会确切地知道，这种情况何时发生——它们与其它边相遇，此时它才会与Voronoi图的其它部分联接起来。

这样，我们就明白了每个基点事件发生之后所引起的变化：海滩线上会出现一条新的弧，同时 Voronoi 图中也会出现一条不断生长的新边。一条新的弧，是否可能以其它的方式出现在海滩线上呢？答案是不可能。

〖引理 7.6〗

只有在发生某个基点事件时，海滩线上才会有新的弧出现。

〖证明〗

假若本引理不成立，即海滩线有可能被原先某抛物线β_j分割。这种情况无外乎两种可能。

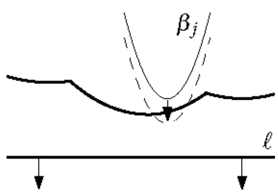


图7-13 假设抛物线β_j的快速生长，并在某时刻“突破”海滩线

前一种可能是，β_j是在由另一条抛物线β_i贡献的某段弧的中间某处，将海滩线分隔开来（如图 7-13 所示）。在要分未分的那一瞬间，β_i和β_j必然是相切的——即，它们相交，而且只相交于一点。将扫描线在此刻的y-坐标记作l_y。若$p_j := (p_{j,x}, p_{j,y})$，则抛物线$\beta_j$的方程就应该是：

$$\beta_j \ := \ y \ = \ \frac{1}{2(p_{j,y} - l_y)} \times (x^2 - 2p_{j,x}x + p_{j,x}^2 + p_{j,y}^2 - l_y^2)$$

当然，β_i的方程也类似。考虑到$p_{j,y}$与$p_{i,y}$均大于l_y不难证明，β_i和β_j的交点不可能仅有一个。因此，原有的任一抛物线β_j，都不可能从另一抛物线β_i贡献的某段弧的中间分割海滩线。

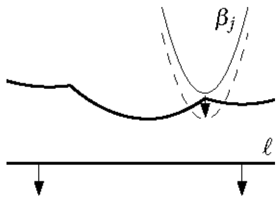


图7-14 抛物线β_j恰好从某对抛物线弧的接合处“突破”

另一可能是：β_j出现于某两段抛物线弧的接合处（图 7-14）。设这两段弧分别来自抛物线β_i和β_k；q为其交点之一，且在该位置β_j即将从海滩线上生长出来。另外假定，沿海滩线，由β_i贡献的那段弧从左侧与q相联，而由β_k贡献的那段弧从右侧与q相联（图 7-15）。此时，这三条抛物线所对应的基点p_i、p_j和p_k，共同确定一个圆C[①]。该圆必与扫描线l相切。从该切点出发沿顺时针方向，各点的次序为：p_i、p_j和p_k——因为，已经假设β_j出现于由β_i和β_k贡献的两段弧的接合处。设想扫描线再下移无穷小的一段，而且此时圆C依然与l相切（图 7-15）。

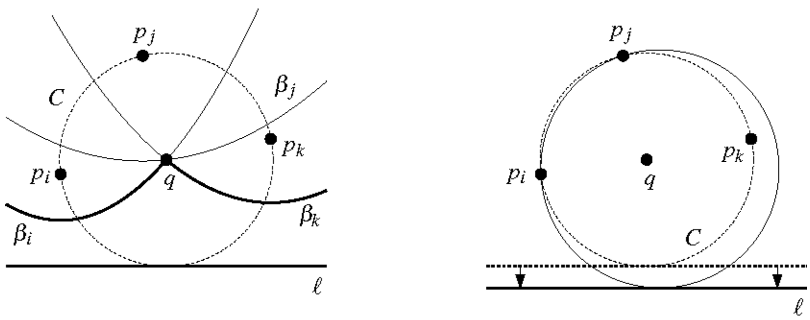


图7-15 β_j在海滩线上出现的瞬间，局部的位置关系以及在扫描线继续前行后对应的圆

此时，若圆C依然通过p_j，其内部就不可能依然是空的——因为，p_i与p_k之一必然会进入该圆的内部。因此，若扫描线继续向下运动，则在q的一个足够小的邻域内，抛物线β_j都不可能在海滩线上出现——因为就其与q[②]的距离而言，此时的p_i与p_k之一必定比p_j更近。 ❑

[①] 其圆心为 q。——译者

[②] 原书此处误作“与 l...”。——译者

由上述引理，立即可以导出一个推论：组成海滩线的抛物线弧，总共不会超过 2n-1 段——只有在遇到一个基点时，才会生出一条新的弧，同时最多将原有的某一条弧一分为二；而其它的时候，海滩线上都不可能会有新的弧出现。

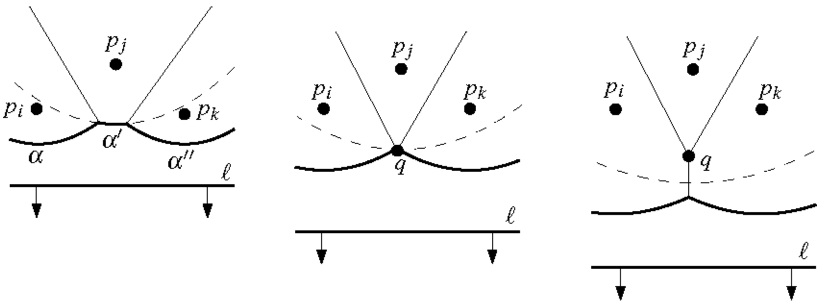


图7-16 海滩线上某段弧的消失过程

平面扫描算法中的第二类事件，发生于原有的某段弧收缩为一点并即将消失时（图 7-16）。设该弧为α'，且它消失之前弧α和α"与之相邻。弧α和α"不可能来自同一条抛物线——仿照〖引理 7.6〗证明中排除前一可能的方法，即可排除这种情况。于是，α、α'和α"这三段弧必然分别对应于三个不同基点p_i、p_j和p_k。就在α'即将消失的那一刻，这三个基点所对应的抛物线将相交于同一点q。此时点q到扫描线l与到这三个基点等距离。亦即，存在一个以q为中心、穿过p_i、p_j和p_k的圆，且该圆在最低点处与l相切。该圆的内部不可能有任何基点——否则，q到该基点将比到l更近，而这却与“q位于海滩线上”的事实不合。因此，点q必是Voronoi图的一个顶点。这很自然，因为先前我们已经注意到，海滩线上各断点所勾勒出的轨迹正是Voronoi图。因此，若海滩线上有某段弧消失，并因而有两段弧汇合起来，则相应地在Voronoi图中肯定也会有两条边汇合起来（成为一条新的边）。海滩线上依次首尾相联的任何三段弧，其对应的三个基点都会确定一个外接圆；当扫描线触及某个这类外接圆的最低点时，也就发生了一次**圆事件**（circle event）。以上分析可得如下引理：

〖引理 7.7〗

海滩线上已有的弧，只有在经过某次圆事件之后，才有可能消失。

至此我们已经弄清楚，什么时候海滩线的组合结构将发生变化，以及如何变化：发生一次基点事件，就会出现一段新弧；发生一次圆事件，已有的某段弧就会消失。此外，我们也知道了它们与待构造出来的 Voronoi 图有何联系：基点事件发生时，就会有一条新的 Voronoi 边出现并朝两端生长；圆事件发生时，会有两条正在生长的边汇合起来，并在接合处形成一个 Voronoi 顶点。剩下的问题是：如何找到一种适宜的数据结构，以维护在扫描过程中所必需的那些信息。既然我们的目标是构造出 Voronoi 图，该结构首先就必须能够将目前已计算出的 Voronoi 图局部记录下来。此外，任一扫描线算法都必备的两个数据结构——事件队列（event queue），以及记录扫描线当前状态的结构——

也是必需的。此处，后一结构所记录的就是海滩线的状态。可按如下方法，实现这些数据结构：

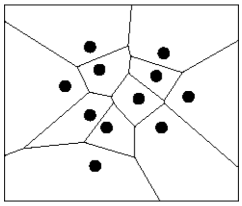


图7-17 为了配合双向链接边表的使用，在算法开始之初，即引入足够大的一个包围框（bounding box）

- 双向链接边表（doubly-connected edge list）是适宜于存储子区域划分的一种通用数据结构，我们可以利用这种结构来存储逐步构造出来的Voronoi图。不过严格地讲，Voronoi图还不是第 2 章中所定义的那种子区域划分——在Voronoi图中，可能有射线，甚至有完整的直线，而这些对象都不能通过双向链接边表来表示。在构造的过程当中，这还算不上是一个问题——因为根据（后面将要介绍的）海滩线的表示方法，在构造过程中，我们可以有效地对这种“双向链接边表”中的相关部分进行操作。然而我们希望，一旦计算结束，最终得到的应该是一个合法的双向链接边表。

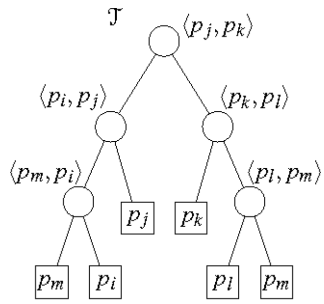


图7-18 用平衡二分查找树存储海滩线对应的状态结构[①]

- 我们借助一棵平衡二分查找树T来存储海滩线，它就是这里的状态结构（status structure）。如 图 7-18 所示，其中的每一匹叶子，分别对应于海滩线上的某段弧。

  由于海滩线是x-单调的，各段弧所对应的叶子必然是有序的：最左边的叶子对应于最左边的弧，接下来的一匹叶子对应于最靠左的第二条弧，依此类推。每匹叶子μ不仅要记录其代表的那段弧，而且也要记录下这段弧所对应的基点。T的内部节点则分别对应于海滩线上的各个断点。每个内部节点所对应的断点，都表示为一个有序的基点对<pi, pj>，其中pi、pj分别对应于交汇于该断点的左、右两条抛物线。若按照这样的方式来表示海滩线，每遇到一个新的基点，都可以在O(logn)时间内，沿海滩线找出位于该基点上方的那段弧：在查

① 第三版遗漏了此图，现借用第二版原图。——译者

找过程中，在每个内部节点处，只要将其对应断点的x-坐标，与新基点的x-坐标做一比较——我们已经知道了定义该断点的一对（抛物线所对应的）基点的位置，同时也知道了扫描线的位置，因此必然可以在常数时间内完成每次比较。请注意，这里并未显式地存储各条抛物线（如 图 7-19 所示）。

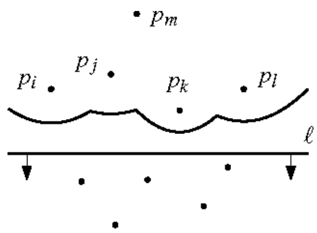


图7-19 沿着海滩线，各段弧所对应的叶子在T中也是按序排列的[①]

在扫描的过程中，还要用到另外两个数据结构，我们还要在 T 中设置指针，指向这两个数据结构。T 的一匹叶子若对应于某段弧α，则为它配备一个指针，指向事件队列中的一个（事件）节点——具体说，就是（在将来可能）导致α消失的那个圆事件所对应的节点。若没有导致α消失的圆事件，或者还没有发现这样一个事件，则该指针被置为 nil。最后，每个内部节点 v 也配有一个指针，指向与当前 Voronoi 图对应的双向链接边表中的某条半边（half-edge）——更确切地说，此时与 v 相对应的断点，正在勾勒出的一条 Voronoi 边，而 v 的指针就指向这条边所对应的那条半边。

- 事件队列 Q 可以用优先队列来实现，其中，各事件的优先级就是其 y-坐标。这个结构记录的，是所有已知的将要发生的事件。对于基点事件，只需记录与之对应的基点。对于圆事件，在对应的事件点处，不仅要记录该圆的最低点，还要设置一个指针指向 T 中的某匹叶子——这匹叶子所对应的，就是在该事件发生时即将随之消失的那段弧。

所有的基点事件都可以在事先确定，而圆事件则不然。这就引出了我们最后需要讨论的一个问题——如何发现圆事件。

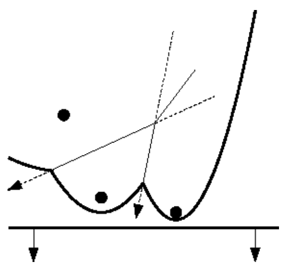


图7-20 邻接弧三元组

---

① 第三版遗漏了此图，现借用第二版原图。——译者

在扫描过程中每发生一次事件，海滩线的拓扑结构都会发生相应变化。如 图 7-20，若将“邻接弧三元组”（a triple of consecutive arc）定义为“沿海滩线依次首尾衔接的任意三段弧”，则此时可能会出现新的三元组，而原有的三元组也可能会消失。任一邻接弧三元组只要确定了一个可能发生的圆事件，算法都必须保证该事件记录到队列 Q 中。这里有两个微妙之处需要说明。首先，某些邻接弧三元组所对应的两个断点，可能不会汇聚到一起——亦即，其各自的移动方向，已经注定了它们永远也不会汇合。若这两个断点分别沿着已经相交过的两条平分线移动，即出现上述情况。此时，该邻接弧三元组根本就不会确定任何可能发生的圆事件。第二点是，即使某个邻接弧三元组所对应的两个断点会逐渐靠拢，其对应的圆事件最终也不见得肯定会发生——可能出现的一种情况是：该事件还没有来得及真正发生，这一邻接弧三元组就已经消失了（比如说，在此之前，海滩线上的这一段会触及到某个新的基点）。这样的一个圆事件，称作误警（false alarm）。

因此算法的实际过程如下：每遇到一个事件，都逐一检查新出现的邻接弧三元组。比如，每一基点事件都会产生三个新的邻接弧三元组——新弧在其中分别居于左侧、中间和右侧。若该邻接弧三元组所对应的两个断点趋于汇合，则将对应的圆事件插入队列 Q。请注意，在基点事件发生时，新弧在其中居中的那个邻接弧三元组，绝不可能引发圆事件——因为，分别居于左、右侧的两段弧必然来自同一抛物线，故此后其对应的两个断点必然会彼此远离。此外，所有从此消失的邻接弧三元组，也要逐一检查，看看在 Q 中是否记录有与之对应的圆事件。若有，则显然是一次误警，故需将其从 Q 中删去。由于 T 的每一匹叶子都设有指针指向 Q 中对应的事件，故该任务很容易完成。

〖引理 7.8〗

每个 Voronoi 顶点，都会在某次圆事件发生时被发现。

〖证明〗

对任一Voronoi顶点q，令 p_i、p_j 和 p_k 为三个基点，它们共同确定了内部不含任何基点的圆 $C(p_i, p_j, p_k)$。由〖定理 7.4〗，该圆及对应的三个基点的确存在。为简明起见，只证明这样一种情况：除 p_i、p_j 和 p_k 外，$C(p_i, p_j, p_k)$ 不经过其它的任何基点，且这三个基点都不是该圆的最低点。不失一般性地，假设从 $C(p_i, p_j, p_k)$ 的最低点起沿顺时针方向遍历（traverse）该圆，将依次经过 p_i、p_j 和 p_k。

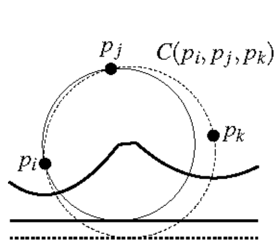


图7-21 扫描线即将触及 $C(p_i, p_j, p_k)$ 最低点的瞬间，p_i、p_j 和 p_k 分别对应于依次首尾相联的三段弧

我们必须证明：就在扫描线即将触及$C(p_i, p_j, p_k)$上最低点的那一刹那，在海滩线上，p_i、p_j和p_k分别对应于依次首尾相联的三段弧α、α'和α"（如 图 7-21 所示）。唯有如此，对应的圆事件才会真正发生。在扫描线触及$C(p_i, p_j, p_k)$的最低点之前，考虑一个无穷接近于这一时刻的时刻。既然$C(p_i, p_j, p_k)$之上及其内部都不含任何其它的基点，必定存在一个经过p_i和p_j、与海滩线相切的圆，而且这个圆的内部也不含任何基点[①]。因此，海滩线上必然有两段相邻的弧，分别对应于p_i和p_j。对称地，海滩线上也必然有两段相邻的弧，分别对应于p_j和p_k。不难看出，这里对应于p_j的"两"段弧，实际上就是同一条。由此可以得出结论：（此时此刻）在海滩线上，p_i、p_j和p_k分别对应于依次首尾相联的三段弧。因此，就在它们对应的圆事件即将真正发生之前，该事件已经被记录在队列$\mathcal{Q}$当中了——而等到（再经过无穷短的时间）该事件真正发生时，对应的Voronoi顶点必然会被发现。 ❑

至此，已经可以给出平面扫描的具体算法。请注意：在所有事件都已处理完毕之后，虽然队列$\mathcal{Q}$已经变空，但是海滩线却依然没有消失。此时，仍然存在若干个断点——它们对应于Voronoi图中的那些单向无穷边[②]。正如前面所指出的，由于并不能用双向链接边表表示这类单向无穷的边，所以必须给整个场景增加一个包围框，然后将这些边与包围框联接起来。算法的整体结构如下：

算法 VORONOIDIAGRAM(P)

输入：平面点集 $P := \{p_1, \ldots, p_n\}$

输出：以双向链接边表$\mathcal{D}$表示的（限制在一个足够大的包围框之内的）Voronoi 图 Vor(P)

1. 初始化事件队列$\mathcal{Q}$：将所有的基点事件插入其中
   初始化状态结构$\mathcal{T}$：将其置空
   初始化双向链接边表$\mathcal{D}$：将其置空
2. **while** ($\mathcal{Q}$ 非空)
3.   **do** 将 y-坐标最大的事件从$\mathcal{Q}$中取出
4.   **if** (这是一个发生于基点 p_i 处的基点事件)
5.   **then** HANDLESITEEVENT(p_i)
6.   **else** HANDLECIRCLEEVENT(γ)
                   (* 这里的γ是$\mathcal{T}$的一匹叶子，它对应于那段即将消失的弧 *)
7. (* 仍然存在于$\mathcal{T}$中的那些内部节点，对应于 Voronoi 图的单向无穷边 *)
   计算出一个包围框，其尺寸之大，应足以容下 Voronoi 图中的所有顶点
   通过对双向链接边表的适当调整，将这些单向无穷边都联接到这个包围框上
8. 遍历双向链接边表中的所有半边

[①] 一般地，经过直线外（同侧的）两个点、与直线相切的圆有两个；只有在这两个点到直线的距离相等时，才只有一个这样的圆。这里取的，是其中无限接近于$C(p_i, p_j, p_k)$的那个。——译者

[②] 更准确地说，应该是"朝下方的单向无穷边"。——译者

增加相应的单元记录
设置好指向这些单元的指针，以及由这些单元发出的（指向对应各边的）指针

处理两类事件的子程序分别如下：

算法 HANDLESITEEVENT(p_i)

1. 若 $\mathcal{T}$ 为空，则
   将 p_i 插入其中（* 这样，$\mathcal{T}$ 只由单匹叶子组成，这匹叶子记录了 p_i 的信息 *）
   然后返回
   否则
   继续执行第 2~5 步
2. 从 $\mathcal{T}$ 中查找出位于 p_i 垂直上方的那段弧 α
   若对应于 α 的那匹叶子有一个指针，指向 $\mathcal{Q}$ 中某个圆事件，则
   (* 说明该圆事件是一次误警 *)
   在 $\mathcal{Q}$ 中剔除该事件
3. 将原来对应于 α 的那匹叶子，替换为一棵由三匹叶子构成的子树：
   居中的那匹叶子记录下新基点 p_i
   两侧的两匹叶子分别记录下原先记录于 α 的基点 p_j
   两个内部节点分别对应于新生出的断点，它们分别记录下基点对 $<p_j, p_i>$ 和 $<p_i, p_j>$
   若有必要，须对 $\mathcal{T}$ 做调整，以使之重新平衡
4. 对应于将 $\mathcal{V}(p_i)$ 和 $\mathcal{V}(p_j)$ 分割开来的那条边，在 Voronoi 图结构中创建一个半边记录
   (* 这条边将由两个新断点逐渐勾勒出来 *)
5. 找出 p_i 在其中居于左侧的那个邻接弧三元组，检查是否有断点汇聚到一点
   果真如此，则
   将对应的圆事件插入事件队列 $\mathcal{Q}$，并
   在 $\mathcal{Q}$ 中该节点和 $\mathcal{T}$ 中与之对应的节点之间设置指针，使它们相互指向对方
   找出 p_i 在其中居于右侧的那个邻接弧三元组，做类似的处理

算法 HANDLECIRCLEEVENT(γ)

1. 将（对应于即将消失的弧 α 的那匹）叶子 γ，从 $\mathcal{T}$ 删除掉
   检查相关的内部节点，更新其中表示有关断点的基点对信息
   若有必要，须对 $\mathcal{T}$ 做调整，以使之重新平衡
   在 $\mathcal{Q}$ 中，删除所有与 α 相关的圆事件
   (* 在 $\mathcal{T}$ 中，γ 的前驱与后继节点配有相应的指针 *)
   (* 借助这些指针，就可以找出这些事件 *)
   （α 在其中居中的那个圆事件，此刻正在接受处理，并已经从 $\mathcal{Q}$ 被删除掉了）
2. 更新存储当前Voronoi图的双向链接边表 $\mathcal{D}$：
   对应于该事件的圆心
   生成一个Voronoi顶点记录，并
   将该记录插入双向链接边表

        对应于海滩线上新生出[①]的断点
            生成两个半边记录[②]，并
            正确地设置好它们相互之间的指针
        将这三个新记录，与同样终止于该Voronoi顶点的其它半边链接起来
3. (* 此前与α紧邻于左侧的那段弧，现可能在某个新的邻接弧三元组中居中 *)
    检查该邻接弧三元组所对应的两个断点是否汇合为一点
    果真如此，则
        将对应的圆事件插入到事件队列 *Q* 中，并
        在 *Q* 中该节点和 *T* 中与之对应的节点之间设置指针，使它们相互指向对方
    (* 此前与α紧邻于右侧的那段弧，现也可能在某个新的邻接弧三元组中居中 *)
    对该弧，做类似的处理。

〖引理 7.9〗

Fortune 算法运行的时间为 *O*(n**log**n)，占用的空间为 *O*(n)。

〖证明〗

对树 *T* 以及事件队列 *Q* 的每一次基本操作（诸如插入、删除一个元素等），均不会超过 *O*(**log**n)时间。对双向链接边表的每一次基本操作，只需要常数的时间。在处理每一个事件时，只需进行常数次这类基本操作，因此，每一事件都能够在 *O*(**log**n)时间内处理完毕。基点事件的数目一目了然，为 n 个。另一方面我们注意到，任何一个圆事件，只要它最后的确发生并接受处理，就会生成 Vor(P)的一个顶点；而所有的误警，迟早都会从 *Q* 中被剔除掉，从而不会真正接受处理。对这些误警的处理，无论是生成它还是删除它的操作，都属于对其它事件处理的一部分，故消耗在这部分事件之上的时间，已经被计入到对应的有效事件名下。所以，真正需要处理的圆事件，不会超过 2n - 5 个。这样，本引理所指出的时间、空间上界（upper bound）得证。 ❑

在给出本节的最终结果之前，需要对退化情况稍做讨论。

本算法自上而下依次处理各事件，故若有两个或更多事件处于同一水平线上（比如两个基点 y 坐标相同），即出现退化情况。若这些事件的x-坐标互异，则按照任意次序来处理，都可解决问题。因此，即便有多个事件y-坐标相同，只要其x-坐标互异，则按照任意次序进行处理，都可消除歧义。然而，有可能算法在一开始就遇到这种情况。比如第二个基点事件的y-坐标与第一个相同，则第二个基点的上方根本还没有任何弧——这时，就需要专门编写代码加以处理。以下再考虑事件位置重合的情况。比如，要是存在四个或者更多个基点共圆，且该圆的内部不含任何基点，就会出现多个

[①] 也可以理解为，原来的两个断点汇合之后，继续（朝新的方向）前进。——译者

[②] 这两条半边，是由两个断点在此处汇合之前所勾勒出的。——译者

位置重合的圆事件。此时，该圆的中心也是Voronoi图的一个顶点。且该顶点的度数至少为 4。当然，也可以专门编写一段代码，来处理这类退化的情况。然而，实际上这并不需要。要是我们听任算法按照随机的次序去处理这些事件，结果又会如何呢？如 图 7-22 所示，算法所生成的，并不是一个 4 度的顶点，而会是两个 3 度的顶点——只不过它们的位置重合，而联接于它们之间的那段边的长度为零。如果有必要，只要在算法结束后再做相应的后处理，即可消除这类退化的边。

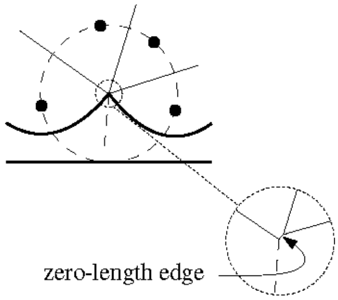


图7-22 多点共圆的退化情况

除这类由于事件的选取次序不同而造成的退化之外，即使是在处理单个事件时也可能会遇到退化的情况。比如，某一基点p_i碰巧位于海滩线上相邻两段弧接合处断点的正下方。如 图 7-23，此时，算法可以在这两段弧之中任选其一，将其一分为二——当然，其中有一段的长度为零——然后把对应于p_i的一段新弧插入于其间。这样，其中长度为零的那段弧，将在对应于一个圆事件的某个邻接弧三元组中居于中间。而该圆的最低点，则与p_i重合。既然该事件的确是由海滩线上三段依次首尾相联的弧确定的，算法就会照例将它插入到事件队列$\mathcal{Q}$当中。等到该事件真正接受处理之时，仍会正确地生成一个Voronoi顶点，而（同时生成的）长度为零的那段弧，同样可以在最后剔除掉。

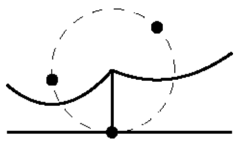


图7-23 海滩线上两段相邻弧之间断点的正下方出现某个基点

还有一种退化情况，其发生的条件是：海滩线上有依次首尾相联的三段弧，分别对应于三个共线的基点。此时，这三个基点不能确定一个圆，故并不会生成任何圆事件。

总而言之，上述算法的确能够正确地应对各种退化情况。

〖定理 7.10〗

给定由平面上任意 n 个基点构成的一个集合，其对应的 Voronoi 图可以采用扫描线算法，在 $O(n\log n)$ 时间内、使用 $O(n)$空间构造出来。

7.3 线段集 Voronoi 图

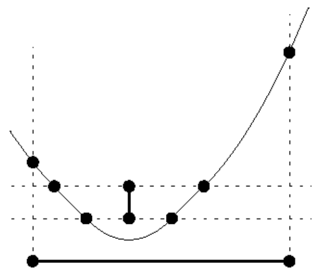


图7-24 一对不相交线段之间的平分线，由不超过七段直线段和抛物线弧组成

点以外的其它物体也可定义 Voronoi 图。此时，平面上一点到物体的距离，定义为该点到物体上各点的最近距离。两点之间的平分线必是一条直线，但不相交线段之间的平分线的形状却要更为复杂。它最多可分为七段，每一段或是直线段，或是抛物线弧（parabolic arc）。若到一条线段的最短距离在其某个端点达到，而到另一条线段的最短距离却是在其内部某点达到，则对应于一条抛物线弧。其它的情况下，都对应于一条直线段。尽管平分线更为复杂，并因此导致 Voronoi 图更为复杂，但对于 n 条互不相交的线段而言，其 Voronoi 图的顶点数、边数以及面数都仍是 $O(n)$量级。

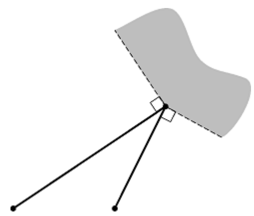


图7-25 若允许线段端点重合，则平分线的定义与计算将十分复杂[①]

倘若仅要求各线段互不穿越，而允许它们共享端点。于是对于这样的一对线段，平面上某一区域中的所有点都将与这两条线段（的公共端点）距离相等，从而导致它们之间的平分"线"不再是曲线。因此，在允许线段端点重合时，为避免在定义与计算线段集 Voronoi 图时遇到此类困难，干脆假定所有线段都严格互不相交。在许多应用中，只需稍微收缩线段，即可保证这一点。

针对点集的扫描线算法，稍作改动后即可应用于线段基点集。设 $S = \{s_1, \ldots, s_n\}$为 n 条互不相交的线段。其中的各条线段依然称作基点（site），而且以下将以基点端点（site endpoint）、基点内部（site internal）指代线段的端点、线段的内部。

此前针对点基点（point site）的算法维护了一条海滩线（beach line）——即由多条抛物线弧联接而成的一条x-单调曲线，该曲线上每一点到扫描线以上所有基点的（最近）距离，等于其到扫描

[①] 第三版原图中，阴影区域的边界误做两条线段的延长线；实际上，应为两条线段在共同端点处各自的垂线。经与作者确认，替换为此图。——译者

线的距离。在引入线段基点之后，新的海滩线又将是什么样子？首先可以看出，一条线段可能部分位于扫描线以上，同时部分位于其下。在定义海滩线时，我们依然仅关注位于扫描线上方的那些基点。对于任一扫描线l，海滩线上各点到l以上各基点的最近距离，仍将等于它到l的距离。这就意味着，此时的海滩线仍由抛物线弧和直线段组成。其中，抛物线弧上的每个点，到某一基点端点的距离最近；直线段上的每个点，到某一基点内部的距离最近。请注意，一旦某一基点内部（如 图 7-26 中的基点s_2）与l相交，则海滩线上必有两条直线段以此交点作为（共同的）端点。

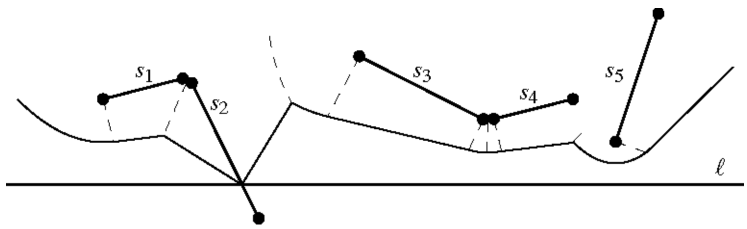


图7-26 一组线段基点（在某一时刻）所对应的海滩线。各断点沿着虚弧线，逐渐勾勒出对应的Voronoi边

海滩线上各抛物线段与直线段之间的断点，分为五种类型。如 图 7-26 所示，为做一详细说明，不妨考虑水平扫描线l向下移动的某一时刻。

1. 若断点 p 到两个基点端点最近，且到它们与到 l 等距，则断点 p 将参与一条直线段的勾勒——此时与点基点（point site）的情况一样。
2. 若断点 p 到两个基点内部最近，且到它们与到 l 等距，则断点 p 将参与一条直线段的勾勒。
3. 若断点 p 到某个基点端点和另一个基点内部最近，且到它们与到 l 等距，则断点 p 将参与一条抛物线弧的勾勒。
4. 若断点 p 到某一基点端点最近，且该距离由该线段基点（segment site）的一条垂线实现，同时该距离等于 p 到 l 的距离，则断点 p 将参与一条直线段的勾勒。
5. 若基点与扫描线相交于内部，则该交点就是一个断点，而且它将参与一条直线段的勾勒。

就第四和第五种情况而言，断点所勾勒的实际上并非 Voronoi 图的一段弧——因为这里仅涉及到一个基点。为保证算法的正确性，有必要对此类断点及其对应的事件进行处理。

与点基点的扫描线算法一样，这里也有基点事件（site event）和圆事件（circle event）。扫描线每触及一个基点端点，就发生一次基点事件。显然，上端点所对应基点事件的处理方法，与下端点有所不同。经过每一上端点后，海滩线的某条弧将被一分为二，同时其间将有四条新弧出现。新弧之间的断点，则都属于后两类。经过每一下端点之后，该基点内部与扫描线之间交点所对应的断点，将被替换为两个第四类的断点，二者之间由（对应于这个新出现的基点端点的）一条抛物线弧联接。

圆事件也类似地分几类。无论哪一类，对应于每一圆事件，海滩线上都会有某条弧消失。扫描

线抵达每一空圆（empty circle）的底部，都会发生一次圆事件——这些空圆均由位于扫描线上方的两或三个基点联合确定。每个空圆的中心，都会有两个相邻的断点汇合。根据汇合断点类型的不同组合，可分若干情形分别处理。若两个断点都属于前三类，则总共会涉及到三个基点。若其中之一属于第四类，则仅涉及到两个基点。在线段基点互不相交的前提下，第五类断点不会影响任何基点。

可见，该算法所构造的 Voronoi 图是由直边和抛物线弧组成的一个子区域划分（subdivision）。那么，它是否依然可以双向链接边表（DCEL）的形式来存储呢？可以，甚至无需对 DCEL 结构做什么调整。每张面（face）都可记录与之对应的基点，从而对于任一半边$\vec{e}$，都可以（借助 IncidentFace($\vec{e}$)和 IncidentFace(Twin($\vec{e}$))）找到以$\vec{e}$所在直线为平分线的那对基点。同时，还可以（借助 Origin($\vec{e}$)和 Origin(Twin($\vec{e}$))）找到由边$\vec{e}$联接的那对顶点，故可以在常数时间内确定任一边的形状。

总之，此时的扫描线算法无非是此前针对点基点（point site）那个算法的扩展，只不过需要分更多情况处理罢了。尽管如此，算法仍只需处理 $O(n)$个事件，每一事件的处理也只需 $O(\log n)$时间。

〖定理 7.11〗

n 条互不相交的线段基点所对应的 Voronoi 图，可以使用 $O(n)$空间、在 $O(n\log n)$时间内构造出来。

线段集Voronoi图的一个应用实例即运动规划（motion planning，参见第 13 章）。假设给定可表示为n条线段的一组障碍物，以及一个机器人$\mathcal{R}$。再假定该机器人可以朝任意方向自由移动，而且可以很好地近似为一个包围圆（enclosing disk）D。假设需要在两个位置之间，为该机器人找出一条无冲突的运动路径（collision-free motion），或者判定无路可通。

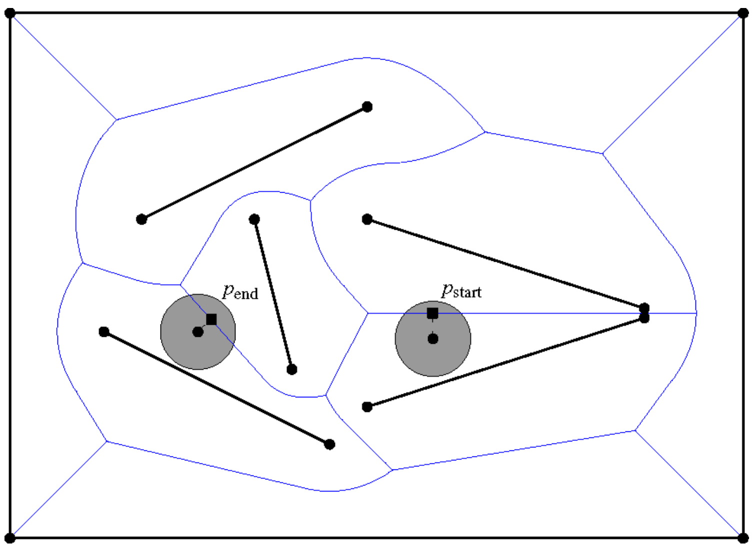


图7-27 与一组线段基点（障碍物）对应的Voronoi图，以及圆盘（机器人）的起始和目标位置

运动规划的技巧之一，就是所谓的收缩（retraction）。其思路是：Voronoi图中的各条弧给出了介于各线段基点之间的中间线，沿这些弧行进遇到障碍的可能性最小，因此其对应的路线也是最好的无冲突运动路径。图 7-27 给出了一个矩形区域内的一组线段，以及与它们相对应的Voronoi图。

采用以下算法，即可在由任意一组线段表示的障碍物之间，规划出一条无冲突的运动路径。

算法 RETRACTION(S, q_{start}, q_{end}, r)

输入：平面上互不相交的一组线段 S := {s_1, ..., s_n};
分别以 q_{start} 和 q_{end} 为中心的两个圆盘 D_{start} 和 D_{end}，
半径为都是 r，而且不与 S 中的任何线段相交。

输出：联接于 q_{start} 和 q_{end} 之间的一条通路，
以其上任一点为中心、以 r 为半径的圆盘，与 S 中的任何线段都不相交。
如果不存在这样的通路，则报告“无通路”。

1. 在一个足够大的包围矩形内，构造 S 的 Voronoi 图 Vor(S)。
2. 分别确定 q_{start} 和 q_{end} 所属的单元。
3. 将 q_{start} 朝 S 中与之最近的那条线段移动，从而在 Vor(S)上确定通路的起点 p_{start}。
   类似地，将 q_{end} 朝 S 中与之最近的那条线段移动，从而在 Vor(S)上确定通路的终点 p_{end}。
   将 p_{start} 和 p_{end} 作为顶点添加到 Vor(S)中，它们将各自所在弧一分为二。
4. 如此改造后的 Voronoi 图中的所有顶点和边，可以当作是一幅图 $\mathcal{G}$。
   若其中有些边到各基点的最短距离不超过 r，则删除之。
5. 采用深度优先搜索（depth-first search）算法，
   在 $\mathcal{G}$ 中确定是否存在从 p_{start} 到 p_{end} 的通路。
   若存在，则报告的通路由三段衔接而成：
   从 q_{start} 到 p_{start} 的线段，$\mathcal{G}$ 中从 p_{start} 到 p_{end} 的通路，加上从 p_{end} 到 q_{end} 的线段；
   否则，报告“不存在通路”。

从 q_{start} 到 p_{start} 的线段不可能有冲突——因为在沿此方向移动的过程中，圆盘到最近障碍物的距离只可能越来越远。同理，从 p_{end} 到 q_{end} 的线段也不可能有冲突。而对于中心已落在 Voronoi 图上的一对圆盘，其间存在一条无冲突的通路，当且仅当沿着 Voronoi 图存在这样的一条通路。因此，对于圆盘形状的机器人，这要存在通路，就必定能够找出来。

〖定理 7.12〗

对于任意 n 条互不相交的线段（障碍物），以及一个圆盘形状的机器人，在机器人的两个位置之间是否存在一条无冲突的通路，可以使用 O(n)空间、在 O(n**log**n)时间内判定。

7.4 最远点 Voronoi 图

现在考察需要 Voronoi 图的另一应用。制造出来的同一种零件，在外形上多少有些差异。如果需要的是完美的圆形零件，就需要对它们做圆度检测（roundness test）。为此要借助坐标度量机（coordinate measurement machine），对零件表面做点采样。假定已经制作出了圆盘，需要测定其圆度（roundness）。用坐标度量机可以在平面上得到圆附近的一个点集 P。所谓点集的圆度，可以定义为包含这些点的最窄圆环的宽度。这里的圆环（annulus），是指介于两个同心圆之间的区域；所谓的圆环宽度（annulus width），即这两个圆的半径之差。

作为最窄的圆环，其内、外边界（圆）必然穿过P中的点。将内、外边界所对应的圆分别记作C_{outer}、C_{inner}，则C_{outer}和C_{inner}必然都会穿过至少一个点——否则，要么进一步收缩C_{outer}，要么进一步膨胀C_{inner}，即可得到更窄的圆环。然而，即使内、外圆都经过一个点，仍不足以得到最窄的圆环。实际上，最窄圆环共有三种情形（如 图 7-28 所示）。无论何种，两个圆所穿过点的总数都是四。

只要内、外圆所穿过点的数目少于上述任一情况，总可以找出更窄的圆环。这里的“找到覆盖指定点集的最窄圆环”问题，在形式上与第 4.7 节所讨论的“找到覆盖指定点集的最小圆盘”问题十分相似。然而，解决最小包围圆（smallest enclosing disc）的方法并不适用于此处的最小包围圆环（smallest enclosing annulus），因为以下性质不再满足：每次新加入的点，若没有落在当前最优圆环之内，则必然被下一最优解的边界穿过。

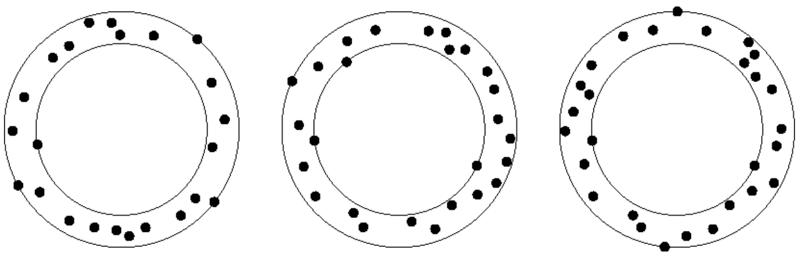


图7-28 最窄圆环的三种可能

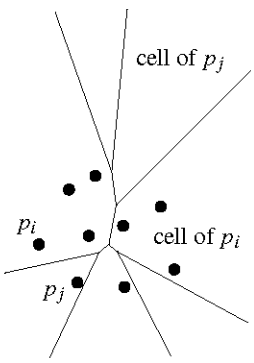


图7-29 最远点Voronoi图的基点与单元

找出最窄圆环，等价于找出其中心点（center point）。实际上，只要中心点（记之为 q）固定了，

最窄圆环也就确定了——其内、外圆分别由距离 q 最近、最远的点确定。如果已构造出该点集的 Voronoi 图，则距离 q 最近的点，也就是其 Voronoi 单元包含 q 的那个点。实际上，最远的那个点也对应于某种几何结构，称作最远点 Voronoi 图（farthest-point Voronoi diagram）——同样地，将平面分割成若干单元，每一单元中的所有点，都距离 P 中的同一个点最远。点 p_i 所对应的最远点 Voronoi 单元，也是 n-1 张半平面的公共交集——这与标准的 Voronoi 图一样，只不过采用的是每条平分线“另一侧”，也就是距离 p_i 更远那一侧的半平面。因此，最远点 Voronoi 图的每个单元都是凸的。不过，在最远点 Voronoi 图中，并非 P 中的所有点都拥有一个单元——因为，此时这些半平面的公共交集可能为空。不难看出，对于平面上的任何一个点，在集合 P 中距离它最远的点，必然位于 P 的凸包上。因此对最远点 Voronoi 图而言，凡是落在凸包内部的点，都不可能拥有一个单元。

〖观察结论 7.13〗

给定平面点集 P，其中任一点在最远点 Voronoi 图中拥有一个单元，当且仅当它是 P 凸包的一个顶点。

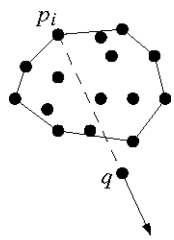


图7-30 最远点Voronoi图中的所有单元都是无界的

还可以证明最远点 Voronoi 图的其它性质。设点 $p_i \in P$ 落在凸包上，且平面上的点 q 距离 p_i 最远。将 p_i 与 q 所在的直线记作 $l(p_i, q)$。考察沿着 $l(p_i, q)$起始于 q、背向 p_i 的那条射线，则该射线上的所有点都会以 p_i 作为最远点，故必落在 p_i 所拥有的单元中。这就意味着，每个单元都是无界的（unbounded）。最远点 Voronoi 图的所有顶点和边（就图的意义而言）组成一个树形结构——它是联通的，同时又不含任何环路。实际上，只要存在环路，就意味着存在有界的（bounded）单元。

习题 7.14 将证明，在n个点的最远点Voronoi图中，顶点、边和单元的数目都是$O(n)$。另一个性质饶有趣味：第 4.7 节中最小包围圆（smallest enclosing disc），其中心或是最远点Voronoi图的顶点，或是贡献了一条Voronoi边的两个基点的中点。前一情形对应于三个最远点，且最远距离相等；后一情形则对应于两个。显然，作为最小包围圆的中心，距其最远的基点不可能只有一个。

最远点 Voronoi 图中同样存在单向无穷边（half-infinite edge），不能直接以双向链接边表（DCEL）形式存储，但对于此类子区域划分，只需稍作调整即可沿用双向链接边表结构。对于没有真正起点的每一条半边（half-edge），我们分别设置一个特殊的顶点记录，作为其（假想）的起点。该记录不再保存坐标，而是代之以单向无穷边的方向。另需要注意的是，在经过上述调整后的 DCEL 中，对于每条单向无穷边所对应的半边$\vec{e}$，要么 Next($\vec{e}$)无定义，要么 Prev($\vec{e}$)无定义。尽管如此，我们不

妨依然称之为“双向链接边表”。

下面介绍一个算法，对于平面上由任意 n 个点组成的点集 P，构造其对应的最远点 Voronoi 图。首先，构造 P 的凸包，保留凸包的顶点，并打乱其次序，记作 $p_1, \ldots, p_h$。然后，将 $p_h, \ldots, p_4$ 从这一循环次序中逐一剔除。剔除 p_i 时，存储其顺时针方向的邻居 $cw(p_i)$、逆时针方向的邻居 $ccw(p_i)$。某个点一经剔除，对于其后被剔除的点而言，它就不再视作为（顺时针或逆时针的）邻居。

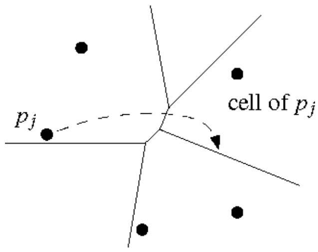


图7-31 每个点p_j都通过指针指向包围其单元的起始边

以 p_1、p_2 和 p_3 的最远点 Voronoi 图为初始的基础，然后采用递增式构造策略。亦即，逐一插入点 $p_4, \ldots, p_n$，不断维护并更新最远点 Voronoi 图。若$\{p_1, \ldots, p_{i-1}\}$对应的最远点 Voronoi 图已构造出来，则为了高效地插入点 p_i 所对应的最远点 Voronoi 单元，需要为每个点 p_j $(1 \leq j < i)$ 配备一个指针，指向双向链接边表中沿逆时针方向遍历（traverse）p_j 的 Voronoi 单元时起始的那条单向无穷边。

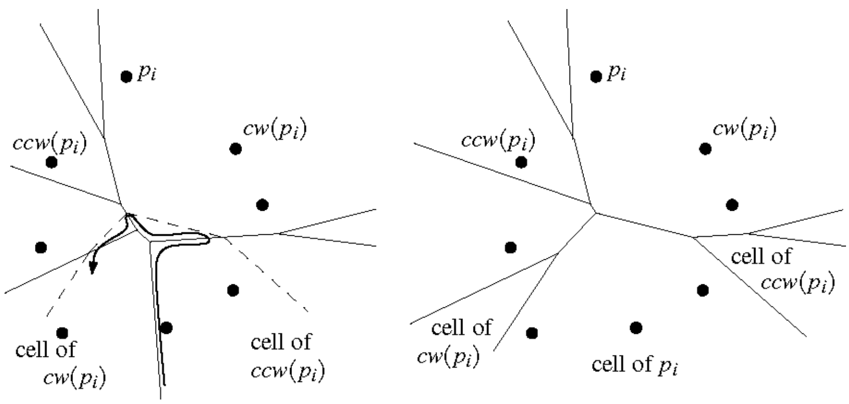


图7-32 将点p_i插入$\{p_1, \ldots, p_{i-1}\}$的最远点Voronoi图中

下面详细介绍插入p_i所对应Voronoi单元的过程，如 图 7-32 所示。该单元必“诞生”于$cw(p_i)$和$ccw(p_i)$的单元之间。在插入p_i之前，沿着$\{p_1, \ldots, p_{i-1}\}$的凸包，$cw(p_i)$和$ccw(p_i)$互为邻居。故其对应的单元必由一条单向无穷边分隔，且该边与$cw(p_i)$和$ccw(p_i)$之间的平分线重合。依前所述，点$ccw(p_i)$必通过一个指针指向该边。此后，沿着p_i与$ccw(p_i)$之间的平分线，将会生出一条单向无穷边，它是p_i与$ccw(p_i)$所对应单元的公共边界。沿顺时针方向遍历$ccw(p_i)$所对应单元的边界，即可确定该平分线与哪条边相交。该边的另一侧，是另一个点p_j（$1 \leq j \leq i-1$）所对应的单元，而p_j与p_i之间的平分线也将为p_i的单元贡献一条边。沿着p_j所对应单元的边界遍历，即可找到下一条平分线以及它贡献的边界。如此，通过顺时针遍历各单元的边界，即可沿逆时针方向依次确定新Voronoi单元的每段边界。最后

一条平分线，必是p_i与$cw(p_i)$之间的那条，在最远点Voronoi图中，它将为p_i的单元贡献另一条单向无穷边。以上所确定的每条新边，都将加至双向链接边表结构中，于是，落在p_i所对应单元内部的所有边都将被删除——在$\{p_1, \ldots, p_i\}$所对应的最远点Voronoi图中，它们已属多余。

概括起来，为完成下一最远点 Voronoi 单元的插入，只需利用当前的 Voronoi 图，顺序追踪新单元（的边界），添加新边，剪除已过时的边。

〖定理 7.14〗

对于平面上的任意 n 个点，都可以使用 O(n)的存储空间，在期望 O(nlogn)时间内构造出其最远点 Voronoi 图。

〖证明〗

找出沿逆时针方向构成凸包的 h 个点，需要 O(n**log**n)时间。一旦确定了按次序构成凸包的这些点，实际上只需 O(h)时间即可构造出最远点 Voronoi 图。可通过后向分析来证明这一点。试考察 p_i 所对应单元刚被插入之后的时刻。可以看出，若 p_i 的单元由 k 条边（半边）围成，则在此前追踪该单元的过程中，应该总共访问了$\{p_1, \ldots, p_{i-1}\}$的最远点 Voronoi 图中的 k 个单元，故而访问到的边不超过 4k-6 条。

$\{p_1, \ldots, p_i\}$的最远点Voronoi图至多包含 2i-3 条边（习题 7.14），每条边都分别被两个单元使用。在$\{p_1, \ldots, p_i\}$中，每个点作为最后一个点被插入的机会是均等的，因此p_i所对应单元的期望规模小于 4。于是，每次插入只需期望的O(1)时间，（在初始化构造凸包之后）算法总共需要期望的O(h)时间。 ❑

现在回到计算最窄圆环（smallest-width annulus）这一问题。假定在该圆环中，C_{inner}至少穿过 P 中的三个点。于是，圆环的中心必是 P 的常规 Voronoi 图中的一个顶点。类似地，若该圆环的 C_{outer} 至少穿过 P 中的三个点，则其中心必是 P 的最远点 Voronoi 图中的一个顶点。最后一种情况，若最窄圆环的 C_{inner} 和 C_{outer} 各自穿过 P 的两个点，则其中心必然既落在常规 Voronoi 图的某条边上，也同时落在最远点 Voronoi 图的某条边上。也就是说，无论何种情形，总是能够在平面上抽取出相当小的一个子集，保证最窄圆环的中心来自其中。

为此，需要对常规 Voronoi 图与最远点 Voronoi 图实施叠合（overlay）运算。所生成新图的所有顶点，正构成了最窄圆环中心的候选集，且它们覆盖了上述三种情况。实际上，并不需要真正地构造出叠合之后的新图。一旦确定了某个顶点，以及定义 C_{inner} 和 C_{outer} 的四个点，即可在 O(1)时间内直接计算出这四个点所对应的最窄圆环——它就是（全局）最窄圆环的一个候选者。

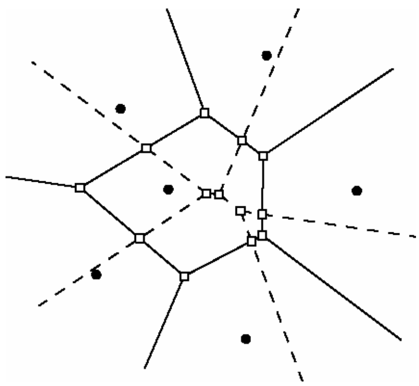


图7-33 常规Voronoi图与最远点Voronoi图的叠合

对于平面上任意 n 点构成的点集 P，计算其最窄圆环的整个算法可概括如下。构造 P 所对应的常规 Voronoi 图与最远点 Voronoi 图。对于最远点 Voronoi 图中的每个顶点，从 P 中找到距其最近的点；对于常规 Voronoi 图中的每个顶点，从 P 中找到距其最远的点。至此可以得到 O(n)个四点组合，它们分别对应于第一或第二种情形中的一个候选圆环。然后，将常规 Voronoi 图的每条边，与最远点 Voronoi 图的各边逐一比对。若相交，则（两条边）对应的四个点确定了一个候选圆环。以上三类候选圆环中的最窄者，即为问题的解。

〖定理 7.15〗

对于平面上的任意 n 个点，都可以使用 O(n)的存储空间，在 $O(n^2)$时间内构造出其最小包围圆环，并由此确定其圆度。

7.5 注释及评论

Voronoi图历史的详细考证，已超出了本书的范围。尽管如此，这里还是不妨做一简要回顾。人们往往将Voronoi图的发现归功于Dirichlet[148]——正因为此，有时也称之为Dirichlet镶嵌（Dirichlet tessellation）——以及Voronoi[379][380]。1644 年出版的《哲学原理》（Principia Philosophiae）的第三篇中，笛卡尔（Descartes）对宇宙爆炸做过分析，那里也出现了这类结构。就在那个世纪，Voronoi图曾数次被重新发现。在生物学领域，这种再发现在极短的时间内就发生过两次。1965 年，Brown[75]曾对树林中的树木密度做过研究。他定义了一个概念，称作“每棵树能够获得的面积”（area potentially available to a tree），实际上，如果把树当作基点，这一概念所指的也就是一棵树所对应的Voronoi单元。一年之后，Mead[272]对一般的植物也采用了这一概念，他将Voronoi单元称为植物多边形（Plant Polygon）。今天，与Voronoi图及其在各研究领域中的应用相关的文献已是浩若烟海。在Okabe等人的著作 [297]中，对Voronoi图及其应用做了详实的介绍。本节所讨论的范围，仅限于在计算几何（computational geometry）的文献中涉及到Voronoi图的方面。

本章证明了Voronoi图的一些性质，其实它还具有更多其它的特性。例如，将相邻Voronoi单元所对应的基点用线段联接起来，就得到了所有基点的一个特殊的三角剖分，称作Delaunay三角剖分（Delaunay triangulation）。这种三角剖分具有很多极好的特性，它也是第 9 章将要讨论的主题。

在 Voronoi 图与凸多面体（Convex Polyhedra）之间，也存在着完美的联系。试考察以下变换：它将 $\mathcal{E}^2$ 中的每个点 $p = (p_x, p_y)$，映射为 $\mathcal{E}^3$ 中的一张非垂直平面 $h(p) : z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$。

如 图 7-34 所示，从几何角度来看，h(p)是单位抛物面$\mathcal{U} : z = x^2 + y^2$ 的某张切平面，其切点的垂直投影正是(p_x, p_y)。对于由平面上若干基点构成的任一点集P，记H(P)为其中所有基点的像平面所构成的集合。现在，考虑由H(P)中各平面所确定的正半空间。所有这些正半空间的公共交集，是一个凸多面体$\mathcal{P}$[①]，也就是说，$\mathcal{P} := \bigcap_{h \in H(P)} h^+$，其中$h^+$表示位于h上方的半空间。奇妙的是，如果将该多面体的顶点及边垂直投影到xy-平面上，就得到了P的Voronoi图 [167]。关于这个变换，在第 11 章中将做详细的介绍。

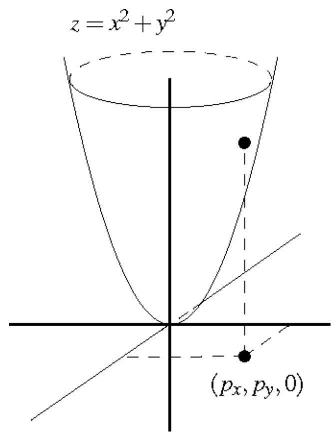


图7-34 通过h(p)变换，在Voronoi图与凸多面体（Convex Polyhedra）之间建立起完美的联系

本章讨论了最基本设置的Voronoi图——也就是说，仅限于欧氏平面上的点集。针对这种条件的第一个最优（$O(n\log n)$）算法，是由Shamos和Hoey[350]提出的，当时他们采用的是一个分治算法。此后，许多不同的最优算法陆续提出。这里所介绍的平面扫描算法，来自Fortune[183]。Fortune最初对本算法的介绍，与此处的讲解有所不同，这里借鉴了Guibas和Stolfi[203]对该算法的理解。

Voronoi图可以从很多方面做推广 [28][297]。其中之一就是，拓展到高维空间中的点集。在$\mathcal{E}^d$中，n个基点所对应的Voronoi图，其最高的组合复杂度（combinatorial complexity，也就是该图中所包含的顶点、边及其它等等的最大总数）为$\theta(n^{\lceil d/2 \rceil})$[239]，而且可以在最优的$O(n\log n + n^{\lceil d/2 \rceil})$时间内被构造出来 [93][133][346]。本章所提及的一些性质，比如“Voronoi图的对偶是所有基点的一个三角剖分”，

① 若将这些平面视为一个排列（arrangement），则该多面体就是这个排列的上包络（upper envelope）。——译者

以及Voronoi图与凸多面体之间的联系等等，在高维空间中依然成立。

另一类推广，涉及到其中所使用的度量。若采用的是 L_1-度量（也称为 Manhattan 度量），则任意两点 p 和 q 之间的距离将定义为：

$$\mathrm{dist}_1(p, q) \quad := \quad |p_x - q_x| + |p_y - q_y|$$

也就是它们的 x-坐标之差、y-坐标之差的绝对值之和。基于 L_1-度量的 Voronoi 图，所有边的方向只有三种可能：水平、垂直或者对角（与坐标轴成 45° 夹角）。若采用的是一般的 L_p-度量，p 和 q 两点之间的距离将定义为：

$$\mathrm{dist}_p(p, q) \quad := \quad \sqrt[p]{|p_x - q_x|^p + |p_y - q_y|^p}$$

请注意，通常的欧氏度量，就是L_2-度量。有好几篇论文 [118][248][252]，对基于这些度量的Voronoi图做过研究。此外，还可以给各基点指派一个权值，并使用权值来定义距离——这样，基点到某个点的距离，不仅要记入它到该点的欧氏距离，还要加上该基点的权值。如此得出的图，被称为加权Voronoi图（Weighted Voronoi diagram）[183]。在定义基点到某个点的距离时，也可以将二者的欧氏距离，乘上基点所具有的权值。按照这种基于乘法的加权距离得到的图，也同样被称作加权Voronoi图[①]（Weighted Voronoi Diagram）[29]。能量图（power diagram）[25][26][27][30]是Voronoi图的另一种推广，其中使用的是一种不同的距离函数（distance function）。甚至，可以完全不考虑任何距离函数，仅仅通过任意两个基点之间的平分线，来定义Voronoi图——这被称为抽象Voronoi图（abstract Voronoi diagram）[240][241][242][274]。

还可以就基点的形状做一般化推广。本章介绍了一组互不相交线段所对应的Voronoi图，并讨论了如何采用收缩（retraction）技巧，将该图用于运动规划（motion planning）。第 13 章将就运动规划做总体介绍。

线段Voronoi图的一个特殊而重要的变种，就是限制于一个简单多边形（simple polygon）内部，关于该多边形各边的Voronoi图。因为相邻的边共享端点，所以在多边形的内部，可能有非零面积的局部与两条边距离相等。实际上，每个凹顶点（reflex vertex）都属于这一情况。此时的Voronoi图，也就是多边形内部的一个子区域划分（subdivision），其中的每一张面，都与一条边或一对边相距最近。这种Voronoi图也被称为中轴（medial axis）或骨架（skeleton）[②]，可用于解决形状分析（shape analysis）等问题。可以在线性正比于多边形所含边数的时间内，构造出它的中轴 [123]。

---

① 无论是加法还是乘法，都有其自然的背景。以树林中的树木为例，在计算“每棵树可能获得的面积”时，如果考虑到不同树木出生的时间不同，则需要通过加法，记入其相应的权值——生日；如果考虑到不同树种在生长速度上的差异，则要通过乘法，记入其相应的权值——生长速度。——译者

② 准确地讲，中轴与骨架并不完全一样，二者之间有细微的差别。——译者

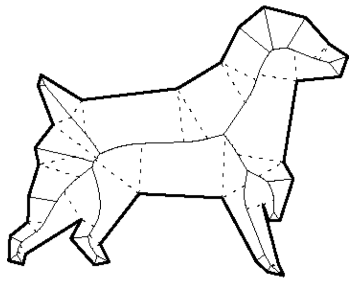


图7-35 多边形的中轴或骨架

通过各点所对应的一个最近基点，可以对整个空间进行子区域划分，而对于某个 $1\leq k\leq n-1$，根据空间中各点所对应的前k个最近基点，也可以得到一种子区域划分。以这种方式得到图被称为高阶Voronoi图（higher-order Voronoi diagram）。对应于特定的k，该图被称为k--阶Voronoi图（Order-k Voronoi Diagram）[6][31][70][98]。请注意，所谓 1-阶Voronoi图，实际上与标准的Voronoi图别无二致。所谓(n-1)-阶Voronoi图也就是最远点Voronoi图（farthest-point Voronoi diagram）——其中，基点p_i所对应的Voronoi单元，就是以p_i为最远基点（的那些点所构成的）子区域。平面上任意n个点所对应的k-阶Voronoi图，其最大的复杂度为$\theta(k(n-k))$[249]。到目前为止构造k-阶Voronoi图的最好算法，运行时间为$O(n\mathbf{log}^3 n + nk)$[6]和$O(n\mathbf{log} n + nk\cdot 2^{c\log^* k})$[326]，其中c为常数。

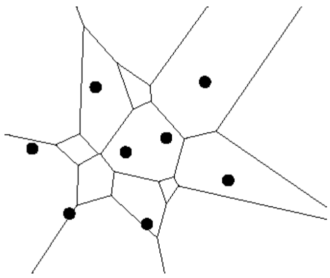


图7-36 高阶Voronoi图

最远点Voronoi图的构造需要$O(n\mathbf{log} n)$时间，不过，如果所有点都出现在凸包上，且它们沿凸包 的排列次序已知，则可采用本章所介绍的简明算法（运行时间为期望$O(n)$）[116]，或者采用$O(n)$的确定算法 [11]。检测物体或点集的圆度（roundness），属于度量衡学（metrology）这一科学度量领域的问题。圆度的定义方法不一而足，本章采取的是最为广泛接受的一种。Ebarra等人 [155]给出过一个平方量级运行时间的算法。Agarwal和Sharir[9]则给出了一个复杂的低于平方时间的算法。针对实践中出现的某些点集特例，还有一些线性时间或接近线性时间的算法 [52][142][187]。Yap和Chang[396]则对计算度量衡学（computational metrology）做过综述。

7.6 习题

习题 7.1　试证明：对于任意的 n > 3，都存在由平面上 n 个基点构成的集合 P，使得 Vor(P)中的某个单元拥有 n - 1 个顶点。

习题 7.2　试证明，由〖定理 7.3〗可以得出一个推论：每个Voronoi单元所拥有的顶点数目，平均不到 6。

习题 7.3　试建立一个从排序（sorting）问题到构造 Voronoi 图问题的归约（reduction），并由此证明：Ω(n**log**n)是后一问题的下界（lower bound）。这里约定：所谓“构造” Voronoi 图的算法，必须能够计算出与该图中每个顶点相关联的所有边，同时得出这些边围绕该顶点的次序。

习题 7.4　根据第 7.2 节中对海滩线及断点的定义，试证明：随着扫描线自上而下扫过整个平面，各断点将勾勒出Voronoi图的所有边。

习题 7.5　试举例说明：某个基点p_i所确定的同一抛物线，有可能为海滩线贡献多段弧。你能否进一步举例说明：其贡献的弧的数目，最大可能达到线性的[1]规模?

习题 7.6　试举例说明：可以找出 6 个基点，使得平面扫描算法只有在处理完它们所对应的 6 个基点事件之后，才会首次遇到圆事件。这 6 点的位置不能是退化的——也就是说，其中任何三点不共线，任何四点不共圆。

习题 7.7　在扫描线向下方推进的过程中，海滩线上的所有断点都是向下方移动吗？试证明这一猜测，或者通过反例证伪。

习题 7.8　试编写一个子程序，在平面扫描结束后，它能够根据尚不完整的双向链接边表以及树 $\mathcal{T}$，构造出一个足够大的包围框。所谓“足够大”，指的是要能够包容下所有的基点和所有的 Voronoi 顶点。

习题 7.9　试编写一个子程序，在找到了一个足够大的包围框之后，它能够在尚不完整的双向链接边表中，加入所有的Voronoi单元记录，并设置好所有相关的指针。——也就是说，将VORONOIDIAGRAM算法中的第 8 行具体化。

习题 7.10　给定由平面上任意n个点构成的一个集合P（如 图 7-37 所示）。



图7-37 最近邻图

试给出一个算法，在 O(n**log**n)时间内找出其中相距最近的一对点。证明你的算法是正

[1] 即 O(n)。——译者

确的。

习题 7.11 给定由平面上任意 n 个点构成的一个集合 P。试给出一个算法，在 $O(n\log n)$时间内，找出与每个点 p 相距最近的点。

习题 7.12 假定已经将点集P的Voronoi图表示为（限制于某一包围框之内的）一个双向链接边表结构。试给出一个算法，在线性正比于实际输出大小的时间内 ①，从P中找出位于其边界上的所有点。

习题 7.13 图 7-26 中共有 10 个断点，请确定它们分别属于五种类型中的哪一种。

习题 7.14 试证明：平面上任意 n 个点的最远点 Voronoi 图，至多包含 $O(2n-3)$条边或半边。其中包含的顶点，又至多有多少？也请就此给出一个确界。

习题 7.15 试证明：无法采用随机增量式算法（randomized incremental algorithm）找到最小包围圆环（smallest enclosing annulus）。为此只需证明：加至P_{i-1}中的某个点p_i，可能并不落在最窄圆环内，同时却又不被$P_i := P_{i-1} \cup \{p_i\}$②的最窄圆环的边界穿过。

习题 7.16 试证明：存在由 n 个点构成的点集 P，其常规 Voronoi 图与最远点 Voronoi 图（各边）之间的交点可多达$\Omega(n^2)$个。

习题 7.17 试证明：若常规 Voronoi 图与最远点 Voronoi 图之间的交点不超过 $O(n)$个，则可以在期望的 $O(n\log n)$时间内确定最窄圆环。

习题 7.18* 在Voronoi分配模型中，客户所希望获得的商品（或服务），其价格在不同基点处都是相同的。假设情况不是这样，在基点p_i处商品的价格为w_i。此时，各基点所对应的商业范围，将对应于它们的加权Voronoi图中的某个单元（参见第 7.5 节）——在这里，每个基点p_i对应的权值就是w_i。试对第 7.2 节中的扫描线算法进行修改，以将其推广到这类情况。

习题 7.19* 假设在对整个平面进行子区域划分之后，得到了n个凸的子区域。我们猜测这个子区域划分就是一张Voronoi图，然而却不知道各基点的具体位置。设计一个算法，如果猜测是正确的，该算法就能找出对应的n个基点 ③。

---

① 即，若 P 中落在其凸包边界上点数为 k，则该算法的时间复杂度为 $O(k)$。换而言之，这是一个输出敏感的（output-sensitive）算法。——译者

② 原书误作“$P_i := P_{i-1} \cap \{p_i\}$”。——译者

③ 若该子区域划分不是一个 Voronoi 图，算法也必须能够检测并报告。——译者

8

排列与对偶：光线跟踪超采样

由计算机生成的三维场景图像，真实感日益提高。与真正的照片相比，如今借助计算机生成的真实感图像已经能够以假乱真了。在这方面的发展中，有一种技术扮演着重要的角色——这就是光线跟踪（ray tracing）。

计算机屏幕由许多称作像素（pixel）的小点组成。高质量屏幕所含的像素，可以多达 1280×1024 个。假设给定一个由若干物体组成的三维场景，外加一个光源和一个视点。为了生成对应于这个场

景的一幅图像，首先需要对屏幕上的每个像素，确定哪个物体[①]对该像素而言是可见的，然后才能确定从该物体上可见的那一点出发、射向视点的那条光线的强度——这个过程也称作绘制（rendering）。我们先来考察前一项工作，即确定与各像素可见的物体。光线跟踪算法在完成这项工作时，要从每个像素发出一条射线（如 图 8-1 所示）。每条光线所碰上的第一个物体，就是与该像素可见的物体。一旦可见的物体已经找到，接下来就可以计算出从这个可见点发出的光线强度。此时，我们必须考虑到这个点从光源接收到了多少光，其中既有直接接收到的，也有通过其它物体的反射而间接接受到的。光线跟踪算法的长处，就在于它能够很好地完成后一项计算任务——而对于图像的真实感来说，这一步计算是至关重要的。不过，本章所要讨论的范围仅限于前一部分的计算。

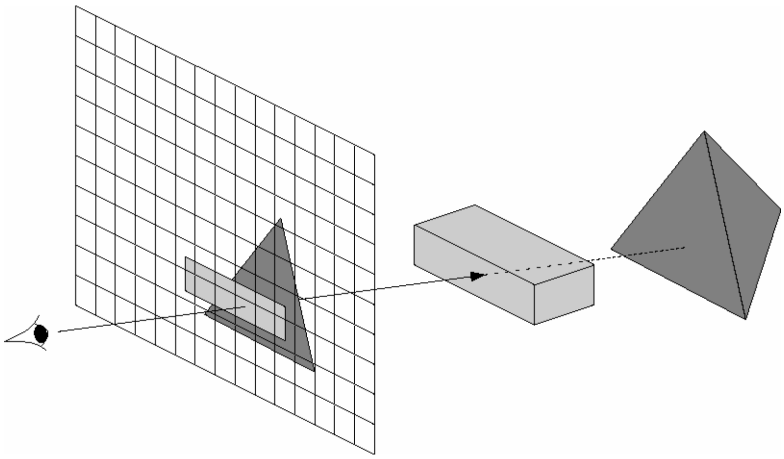


图8-1 利用光线跟踪技术来确定可见的物体

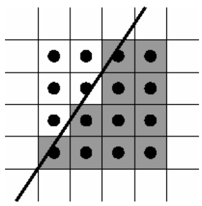


图8-2 图像走样

关于为每个像素确定与之可见的物体，有一个隐藏的问题：所谓的像素，并不是一个点，而是一块很小的方形区域。通常情况下，这并不是什么问题。由于大多数的像素都被某个物体完全覆盖，所以只要从这些像素的中心发出一条光线，就可以确定与之可见的到底是哪个物体。然而，在物体的边界附近，这一问题就会暴露出来。如 图 8-2 所示，当某个物体的一条（直线）边穿过某个像素时，即使这个像素 49%的面积已经被该物体覆盖，从它的中心发出的光线仍然有可能不会穿过这个物体。反过来，若物体覆盖了该像素 51%的面积，这条光线必然会穿过该物体——然而此时我们却

[①] 这里考虑的是最简单的情况，即物体都是完全不透明的。否则，可能会有多个物体对同一像素可见。——译者

会误以为，整个像素都被该物体覆盖了。

图像中屡见不鲜的锯齿现象，根源正在于此。只要还是（将光线与像素相交的结果）生硬地划分为“完全穿过”、“完全错过”两类，就必然会造成这种走样（artifact）。为消除这一问题，应该允许中间情况的出现，比如“49%穿过”。这样就可以根据物体的亮度以及穿过的比率，来设置像素的亮度（比如，像素的亮度等于物体亮度的 49%）。另外，如果某个像素内部（的不同部分分别）与多个物体可见，还可以将这些物体各自贡献的亮度混合起来，做为该像素的亮度。

那么，在光线跟踪算法中，到底应该如何将这些不同的亮度合成起来呢？解决的方法是，从每个像素同时发出多条光线。比如，要是从某个像素发出 100 条光线，其中的 35 条与某个物体相交，那么我们就可以估计出，此物体与该像素的 35%可见。这就是所谓的超采样（supersampling）——在每个像素内的采样点不止一个，而是多个。

为了使这种方法可行，这些光线在每个像素内部应该如何分布呢？一种显而易见的方法就是，将这些光线规则地分布于像素内部。比如，若采用 100 条光线，则它们对应的采样点在像素内部构成一个 10×10 的规则网格。按照这种方法，只要其中的 35 条光线与某物体相交，则该像素内也差不多有 35%的部分与该物体实际可见（如 图 8-3 所示）。

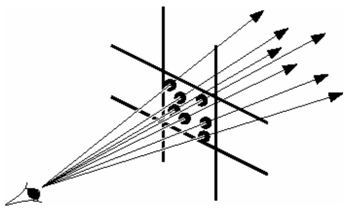


图8-3 对像素的细分

然而，按规则分布进行采样也有另一缺点——尽管对特定像素而言，这种误差的确很小，但是在不同行（或列）的像素之间，却可能由此导致（亮度的）某种规则性。误差的这种规则性分布，将会使人类视觉系统（human visual system - HVS）发生作用，从而令人“看到”不悦的走样。就此而言，按规则分布进行采样并非好主意。更好的办法是按某种随机模式选取采样点。当然，各种随机模式的效果也不尽相同；不过我们依然要求，无论采样点按照何种方式分布，其中（与物体）相交的光线条数（的比率），必须与（该像素）实际被覆盖的面积比率接近。

现在，假设已经生成了一组采样点，需要用某种准则来评价其优劣。一种判断的准则是：光线与某一物体相交的比率，必须与该像素内与此物体可见的面积比率接近。这两个比率之间的差异，就是所谓的“样本集相对于该物体的差异”（discrepancy of the sample set with respect to the object）。当然，事先不可能知道哪些物体会与某个像素可见，因此不得不做好最坏情况的打算。为此，必须考虑一个物体与某个像素的内部可见的所有可能方式，并考虑其中的最大差异——我们的目的，只能是尽量地缩小这个最大差异。这个最大差异，就是所谓的“样本集差异”（discrepancy of the sample

set），这一数值取决于组成场景的物体类型。因此，若是按照正规的形式，就应该相对于给定的某一组物体来定义样本集差异。这样，根据“样本集差异”这一指标，对于任一给定的样本集，我们都可以判断它是否足够好——若差异值（discrepancy）足够小，就保留它；否则，重新选择一组随机的采样点。为此，我们就需要借助于某种算法，来计算出任意点集的差异值。

8.1 差异值的计算

前面曾提到，所谓“点集的差异值”是相对于某一类物体而言的。待考虑的物体就是场景中各三维物体的投影。按照图形学的惯例，这里假定所有弯曲的物体都可近似为多边形网格。这样，需要考虑的二维物体就是多面体（polyhedron）的各小平面（facet）的投影。亦即，我们只是相对于这类多边形来讨论差异值。如 图 8-4 所示，对于一般性的场景，除非包含大量极其狭长或者体积极小的多边形，否则对于任一多边形，多数像素都不会被同一多边形的（至少）两条边同时穿过。

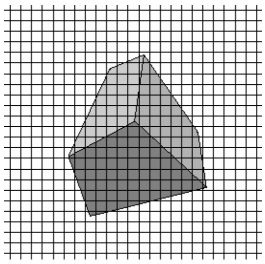


图8-4 一般场景中物体的投影

如果某个像素只与多边形的一条边相交，那么在这个像素内部，这个多边形的作用就相当于一张半平面。像素与多边形的多边同时相交的情况，更为少见。而且，（即使出现这种情况，）也不会导致具有规律性的误差——令人讨厌的走样，正是由于这种规则分布的误差而造成的。因此，我们只需集中精力解决半平面的差异值问题。

设 $U := [0 : 1] \times [0 : 1]$为单位正方形（U 就是像素的模型），由 U 中任意 n 个采样点组成的集合记作 S。考虑由所有可能的闭半平面（closed half-plane）组成的（无穷）集合，记这个集合为 $\mathcal{H}$。对任一半平面 $h \in \mathcal{H}$，h 的连续测度（continuous measure）被定义为集合 $h \cap U$ 的面积。我们记之为$\mu(h)$。比如，要是某张半平面 h 完全覆盖了 U，就有$\mu(h) = 1$。我们也可以定义出 h 的离散测度（discrete measure）。所谓 h 的离散测度，就是包含于 h 中的采样点在所有采样点中所占的比率。若记之为$\mu_s(h)$，则$\mu_s(h) := card(S \cap h) / card(S)$，这里的card(•)表示集合的基数（cardinality）。所谓“h 相对于样本集 S 的差异值”（记作$\Delta_s(h)$），就是连续测度与离散测度之差的绝对值：

$$\Delta_s(h) \ := \ |\mu(h) - \mu_s(h)|$$

例如对于 图 8-5 中的半平面，上述差异值等于 $|0.25 - 0.3| = 0.05$。最后，我们还要定义S的“半

平面差异值”（half-plane discrepancy），也就是所有可能的半平面的差异值的上确界：

$$\Delta_{\mathcal{H}}(S) := \sup_{h \in \mathcal{H}} \Delta_S(h)$$

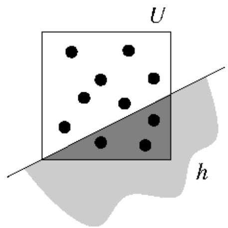


图8-5 连续测度与离散测度之差

如此就定义好了我们所要计算的东西。现在来看看，应该如何来具体地计算它。

所有闭半平面的差异值的上确界，等于所有开半平面、闭半平面的差异值中的最大值。在开始对差异值最大的半平面进行查找之前，首先要确定一个有限的候选半平面集合——将一个无限的候选对象集合替换为一个有限集，总是不错的构思。当然，其前提是，该有限集合依然包含我们感兴趣的那些元素。这里所确定的有限集，就应该包含具有最大差异值的那张半平面。选中的半平面，都是差异值为局部极大的。亦即，无论如何平移或旋转这些半平面，只要平移和旋转的量足够小，差异值总会有所降低。而在这些差异值局部极大的半平面中，必然有某一个达到全局最大差异值。

只要某张半平面的边界线没有穿过 S 中的任何点，就必然可将其轻微平移，使其连续测度有所增长，而其离散测度保持不变。另外，也可将其反向轻微平移，使其连续测度有所降低，而其离散测度保持不变。因此，在这两个平移中，必有其一会使差异值上升。而我们所寻找的（差异值为全局最大的）那张半平面，边界必然通过 S 中的（至少）一个点。现在，考察边界线穿过且只穿过一个点 $p \in S$ 的某张半平面 h。通过围绕 p 旋转 h，是否总能使得 h 的差异值进一步上升呢？亦即，具有全局最大差异值的那张半平面，边界是否必然经过 S 中至少两个点呢？答案为否——在围绕 p 旋转 h 时，可能遇到一个连续测度函数的局部极值。不妨假设是一个局部的最大值。于是，任何足够小的旋转，都将使该连续测度下降。在此局部的最大值处，若离散测度小于连续测度，则这种旋转必使差异值下降。类似地，在连续测度的任一局部最小值处，若离散测度要大于连续测度，则任何轻微的旋转，也必将令差异值下降。因此，全局最大的差异值必出现在某个这类极值点处。

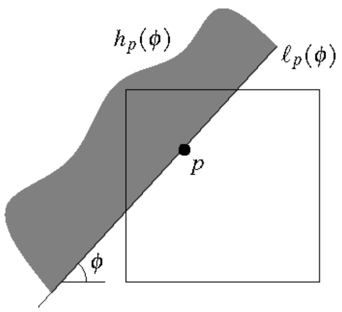


图8-6 全局差异值的极值

让我们来更加仔细地考察这些极值点。如 图 8-6 所示，任取S中的一个点$p := (p_x, p_y)$。对于任一角度 $0 \le \phi < 2\pi$，令$l_p(\phi)$为穿过p、与x-坐标轴正向成ϕ角度的那条直线。$l_p(\phi)$在其上方定义了一张初始的半平面，我们将这张半平面记作$h_p(\phi)$；现考察这张半平面所对应的连续测度。我们所感兴趣的，是函数$\phi \mapsto \mu(h_p(\phi))$的局部极值点。随着$\phi$从 0 到 2π的变化，直线$l_p(\phi)$将围绕p旋转（一周）。首先，当$l_p(\phi)$扫过U的某个顶点时，有可能会出现一个极值点。这种情况最多可能发生 8 次。在两次这种事件之间，$l_p(\phi)$会与U的两条固定边相交。稍做计算，就可以得出所发生的各种情况对应的连续测度。例如，要是$l_p(\phi)$与U的上边界和左边界相交，则有：

$$\mu(h_p(\phi)) = \frac{1}{2}(1 - p_y + p_y \tan\phi)(p_x + \frac{1 - p_y}{\tan\phi})$$

在这种情况下，局部的极值点至多不过两个。若 $l_p(\phi)$与 U 的另外两条边相交时，连续测度函数也有类似的规律。由此可以归纳出一条结论：对于任何点 $p \in S$，局部极值点只有常数个。这样，其边界经过某一指定点的候选半平面，总共只有 $O(n)$个。而且，对于每个点，我们都可以在 $O(1)$时间内，计算出所有极值点以及它们各自对应的半平面。这样就证明了如下引理：

〖引理 8.1〗

设 S 为单位正方形 U 中的 n 个点。相对 S 的差异值达到最大的那张半平面 h，必是以下几种类型：

(i) h 的边界经过某个点 $p \in S$；

(ii) h 的边界同时经过 S 中的两个甚至更多个点。

类型(i)的选半平面的数目为 $O(n)$，而且可以在 $O(n)$时间内找出它们。

显然，类型(ii)候选半平面的数目是平方量级的。既然类型(i)候选半平面的数目远低于平方量级，不妨采用蛮力的（brute-force）方式来处理——这 $O(n)$张半平面中的每一张，都可在常数时间内计算其连续测度，再用 $O(n)$时间计算其离散测度。如此，可在 $O(n^2)$时间内从中找出差异值最大的半平面。在计算类型(ii)候选半平面的离散测度时，需要格外细心，为此需要一些新的技术。本章后续部分将逐一介绍这些技术，并说明如何借助它们在 $O(n^2)$时间内计算所有的离散测度。然后，可以计算所有半平面的差异值（每张半平面只需常数的时间），进而从中找出最大者。最后，只要将这个最大值

与类型(i)候选半平面中差异值的最大值做一比较，即可得到 S 的差异值。故此可得如下定理：

〖定理 8.2〗

在单位正方形内任选 n 个点组成集合 S。都可以在 $O(n^2)$时间内，计算出 S 的半平面差异值。

8.2 对偶变换

平面上的任何一点，都拥有两个参数——x-坐标和 y-坐标。平面上任何一条（非垂直的）直线，也拥有两个参数——其斜率，以及它与 y-坐标轴的交点。因此，可以通过某种一一对应的方式，将一组点映射为一组直线，反之亦然。如果做得巧妙的话，甚至可以将原先点集所具有的某些性质，转换为直线集所具有的某些性质。比如，原先共线的三个点，将映射为共点的三条直线。有多种不同的映射方式，都能做到这一点——它们被称为对偶变换（duality transform）。

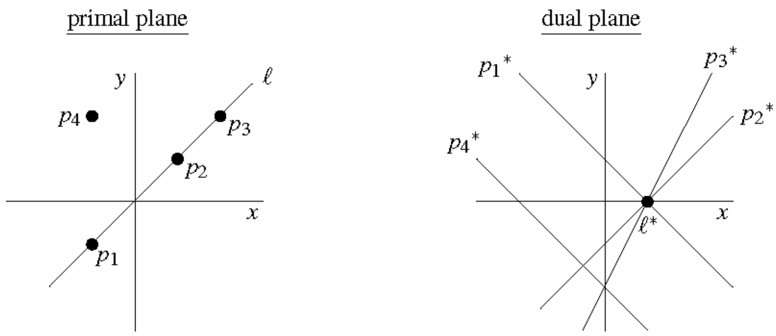


图8-7 对偶变换的一个例子

某个对象经过对偶变换后得到的映射，称为该对象的对偶（dual）。有一种简单的对偶变换定义如下。任意给定平面上的一个点 $p := (p_x, p_y)$。p 的对偶（记作 p^*）定义为：

$$p^* \ := \ (y = p_x x - p_y)$$

而直线 $l : y = mx + b$ 的对偶，就是满足 $p^* = l$ 的那个点 p。换而言之，

$$l^* \ := \ (m, -b)$$

对于垂线，不能定义对偶变换。在大多数情况下，垂线都可以单独进行处理，因此这一点不成问题。当然，还可以通过其它方法来处理垂线——比如，将整个场景作轻微的旋转，从而使得其中不再含有垂线。

我们说，对偶变换将对象从原平面（primal plane）中映射到对偶平面（dual plane）。原来在原

平面中具有的某些性质，在对偶平面中依然成立：

〖观察结论 8.3〗

设 p 为平面上的一个点，l 为平面上一条非垂直线。则对偶变换 $o \mapsto o^*$满足下列性质：

A) 关联性的保持：$p \in l$ 当且仅当 $l^* \in p^*$；

B) （位置）次序的保持：p 位于 l 的上方，当且仅当 l^*位于 p^*的上方。

从 图 8-7 中可以看出这些性质：在原平面中，点p_1、p_2和p_3都落在直线l上；而在对偶平面中，直线p_1^*、p_2^*和p_3^*都经过点l^*。在原平面中，点p_4 位于直线l的上方；而在对偶平面中，点l^*位于直线p_4^*的上方。

除点和直线之外，对偶变换也可以应用于其它的对象。比如，线段 $s := \overline{pq}$的对偶是什么？一种合乎逻辑的方法，就是将 s^*定义为“s 上所有点的对偶的并集”。这样，就得到了一个无穷的直线集。既然 s 上的那些点都是共线的，故与它们对偶的直线必然穿过同一点。这些直线的并集，形成一个双楔形（double wedge），而构成其边界的两条直线，分别是 s 的两个端点的对偶。当然，s 的两个端点所对应的那两条对偶直线，实际上可以定义出两个双楔形——一个是左右式的，另一个是上下式的。这里的 s^*是其中左右式的那个双楔形。

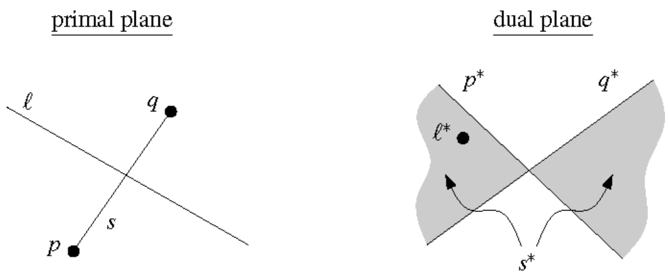


图8-8 将对偶变换应用于直线段

图 8-8 所示的就是一条线段s的对偶。其中还画出了一条与s相交的直线l，可以看到，与它对偶的点l^*落在双楔形s^*内。这并不是一次巧合——相对于与s相交的任何直线，s的端点p和q中肯定各有一个落在上方和下方，因此，根据对偶变换的保序性，与这条直线对偶的点必然落在s^*内。

上面引入的对偶变换，有一个漂亮的几何解释。如 图 8-9 所示，令$\mathcal{U}$为抛物线$\mathcal{U} : y = x^2 / 2$。首先来考察$\mathcal{U}$上任一点p的对偶。在点p处，$\mathcal{U}$的微分是$p_x$，也就是说，$p^*$的斜率与$\mathcal{U}$在点p处的切线相同。而事实上，任一点$p \in \mathcal{U}$的对偶，正是$\mathcal{U}$在点p处的切线——因为，这条切线与y-坐标轴交于$(0, -p_x^2/2)$。现在，再来考虑不落在$\mathcal{U}$上的任一点q。直线$q^*$的斜率是多少呢？请注意，位于同一条垂线上的任意两点，其对偶的两条直线必然斜率相等。在这里，也就是说q^*必与p^*平行——其中的p，就是落在$\mathcal{U}$上、与q的x-坐标相同的那个点。

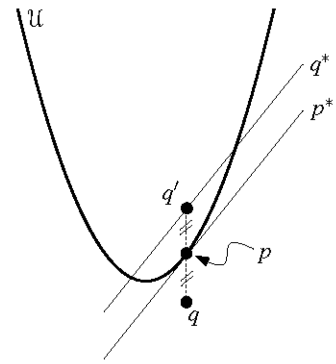


图8-9 对偶变换的几何解释

考虑另一个点 q'，这个点与 q（以及 p）的 x-坐标相同，而且满足 $q'_y - p_y = p_y - q_y$。对于 x-坐标相同的两个点，与它们分别对偶的两条直线之间的垂直距离，等于这两个点的 y-坐标之差。因此，直线 q^*必然穿过点 q'，而且平行于 $\mathcal{U}$ 在点 p 处的切线。

在审视对偶变换之后，或许你会感觉疑惑——它究竟有何用途？实际上，只要你能够在对偶平面中解决某个问题，也就同时解决了在原平面中与之对应的那个问题——你需要做的，仅仅是模仿对偶问题的答案，写出该问题的解。就其本质而言，原问题与对偶问题毕竟是相互等价的。此外，将原问题转化到对偶平面中，还有另一个重要的好处——可以为我们提供一种新的视角。在求解问题时，如果能够从不同的角度来考虑它，往往能够获得更深刻的理解，并由此解决它。

现在就来看看，如果把差异值问题置于对偶平面中加以考虑，将会是什么情况。前一节还有一个问题尚未解决：给定由 n 个点构成的一个集合 S，其中任意两个点都确定一条直线，每条直线又进而界定出两张半平面，如何计算出所有这些半平面各自的离散测度？

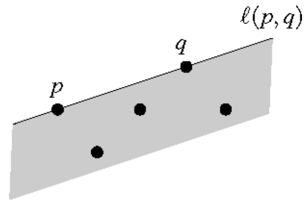


图8-10 原平面：l(p,q)及其下方点

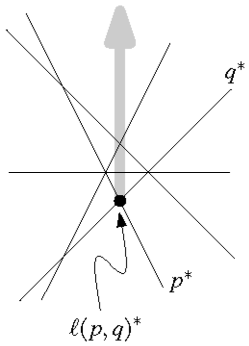


图8-11 对偶平面：：l(p,q)*及其上方的直线

对集合S进行对偶变换后，可以得到一个直线集$S^* := \{p^* : p \in S\}$。如 图 8-10 所示，对任意两点$p, q \in S$，我们将它们所确定的直线记作l(p, q)。若 图 8-11 所示，这条直线的对偶，就是直线$p^*, q^* \in S^*$的交点。考察由l(p, q)界定、位于l(p, q)下方的那张开半平面。这张半平面的离散测度，也就是严格位于l(p, q)下方的点的个数。这就是说，在对偶平面中，我们感兴趣的是严格位于点$l(p, q)^*$上方的（对偶）直线的条数。若换成是位于l(p, q)下方的闭半平面，则还需要记入经过点$l(p, q)^*$的（两条）直线。类似地，对于由l(p, q)界定、位于l(p, q)上方的那张开半平面，我们所感兴趣的就是（严格）位于点$l(p, q)^*$下方的（对偶）直线的条数。

下一节将对直线集做一研究，并给出一个算法，有效地统计出分别位于每个交点上方、下方以及经过每个交点的直线的条数。只要将这一算法应用于 S^*，就可以得到很多信息，这些信息足以帮助我们计算出，穿过 S 中任意两点的各条直线所界定的半平面所对应的离散测度各是多少。

有一个问题需要仔细处理：x-坐标相同的任意两点，必然对偶于斜率相等的两条直线。因此，与这两点所确定的那条直线“对偶”的交点，在对偶平面中并不会出现。这不足为奇，因为对于垂线，对偶变换本来就没有定义。在具体的应用中，需要对此做一步附加的处理。对于穿过至少两个点的每条直线，我们都必须计算出其所对应半平面的离散测度。经过 S 中两个（或者更多）点的垂线，数目不会超过线性的规模，因此，即使是使用蛮力的方法，也可以在总共 $O(n^2)$时间内，计算出所有这类直线对应的离散测度。

8.3 直线的排列

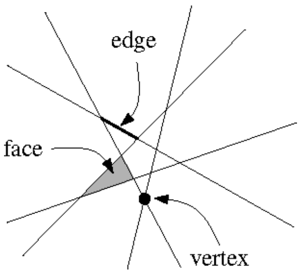


图8-12 直线的排列

如 图 8-12 所示，设L为由平面上n条直线组成的集合。由集合L，可以导出平面的一个子区域划分（subdividsion），这个子区域划分由顶点、边以及面组成。其中有些边和面是无界的。这种子区域划分，通常被称为“由L导出的排列（arrangement）”，记作*A*(L)。如果其中任何三条直线都不共点，而且任何两条直线都不平行，我们就称之为“简单的”（simple）。

所谓一个排列的（组合）复杂度，就是其中所有顶点、边和面的总数。无论是由直线构成的排列，还是其在更高维空间中的推广，在计算几何（computational geometry）中都会经常遇到。在一个点集上定义的一个问题，经过对偶变换之后，往往可以转化为另一个关于排列的问题。之所以这样做，是因为较之点集的结构，直线排列的结构更加直观易见。例如，在原平面上经过两个点的一条直线，在对偶排列中将变成一个顶点——这样，这一特性就会变得更加清楚明确。排列的附加结构并不是没有代价的。事实上，构造一个完整的排列（结构）不仅费时，而且（得出的结构）也会占用大量的空间——毕竟，排列的组合复杂度（combinatorial complexity）是很高的。

〖定理 8.4〗

设 L 为由平面上 n 条直线构成的任一集合，而 *A*(L)为由 L 导出的排列。则

(i) *A*(L)中的顶点不超过 n(n-1)/2 个；

(ii) *A*(L)中的边不超过 n^2 条；

(iii) *A*(L)中的面不超过 $n^2/2 + n/2 + 1$ 张。

上述命题中的等号成立，当且仅当 *A*(L)是简单的。

〖证明〗

A(L)中的每一个顶点，都必然是 L 中某（至少）两条直线的交点。因此，其总数至多为n(n-1)/2。要出现这样多个顶点，当且仅当每一对直线都能贡献一个交点，而且不同的直线对所贡献的交点互不相同——也就是说，*A*(L)是简单的。

在任何一条直线上，边的条数要比顶点的个数正好多一。后一数目至多为n-1，故任何一条直线所贡献的边不会超过n条。所有直线累计起来，边的总数至多为n^2。要出现这样多条边，

当且仅当$\mathcal{A}(L)$是简单的[①]。

为了得出$\mathcal{A}(L)$中所含面数的上界（upper bound），可以依次引入各条直线，并估计出每一次所增加面数的上界。令 $L := \{l_1, \cdots, l_n\}$。对每一个 $1 \le i \le n$，定义 $L_i := \{l_1, \cdots, l_i\}$。在引入 l_i 后，$\mathcal{A}(L_{i-1})$将变成$\mathcal{A}(L_i)$，而在此过程中，会增加多少张面呢？考虑由 l_i 贡献的各条边——其中每条边都将原来$\mathcal{A}(L_{i-1})$中的某张面一分为二。因此，新增加的面数，正好等于 l_i 在$\mathcal{A}(L_{i-1})$中被分割成的段数——这个数至多为 i。因此，面的总数至多为

$$1 + \sum_{i=1}^{n} i = n^2/2 + n/2 + 1$$

同样，要出现这样多张面，当且仅当$\mathcal{A}(L)$是简单的。 ❑

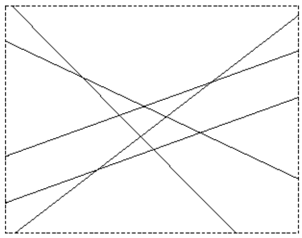


图8-13 包围框（bounding box）

因此，由直线集L导出的排列，是复杂度不超过平方量级的一个平面子区域划分。乍看起来，双向链接边表（doubly-connected edge list）似乎适宜于存储排列结构——以这种表示方式，可以有效地枚举出围成某一给定面的各边，也可以从一张面转到与之紧邻的下一张面，等等。然而，双向链接边表中只能存放有界的边，而任何一个（非空）排列中总有一定数量的无界边。所以，如 图 8-13 所示，需要引入一个足够大的包围框，将排列中的所有重要部分都包含进去——亦即，必须将排列中的所有顶点都包含进去。现在，这个包围框以及原排列落在包围框中的部分,共同构成了一个子区域划分，而这个子区域划分中的所有边都是有界的，这样才可以用一个双向链接边表来存储。

那么，这个双向链接边表又应如何构造呢？在我们脑子里首先闪过的，就是平面扫描算法。第 2 章曾借助平面扫描算法，计算一组线段之间的所有交点；后来，在计算两个平面子区域划分经叠合之后所得的双向链接边表时，这种算法也派上过用场。实际上，采用第 2 章那些算法构造排列$\mathcal{A}(L)$并不困难。然而，这里的交点数目是平方量级的，故算法的运行时间将是$O(n^2\log n)$。这个性能虽不算太差，但也不是最优。因此，我们转而尝试脑子里闪过的下一个方法——递增式构造算法。

很容易就可以在平方量级的时间内构造出一个包围框 $\mathcal{B}(L)$，将 $\mathcal{A}(L)$的所有顶点包含进去：计算出各直线对之间的所有交点，并从中找出最靠左侧的、最靠右侧的、最靠下的和最靠上的。构造出方向与坐标轴平行、包含这四个点的一个矩形，该矩形必然同时包含了该排列中的所有顶点。

[①] 亦即，任何两条直线都相交，而且所有交点互异。——译者

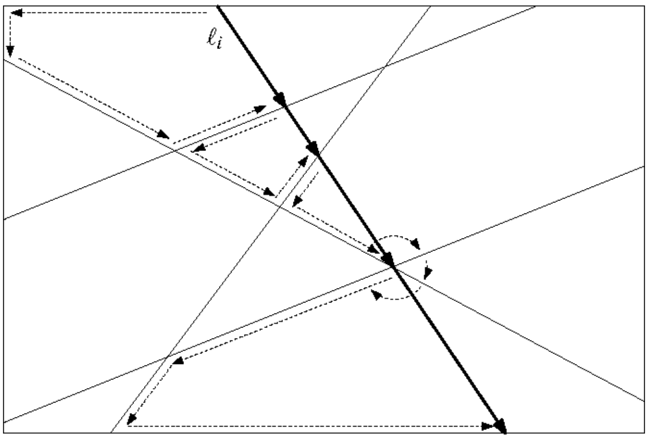


图8-14 遍历一个排列

按照递增式算法，直线$l_1, \ldots, l_n$将逐一引入，每引入一条新的直线，都要相应地更新双向链接边表。包围框$\mathcal{B}$(L)以及$\mathcal{A}$({$l_1, \ldots, l_i$}限制于$\mathcal{B}$(L)之内的部分，共同导出了一个平面子区域划分，记作$\mathcal{A}_i$。在引入l_i时，需要在$\mathcal{A}_{i-1}$中找出与l_i相交的那些面，并分别将它们一分为二。为了找出这些面，可以如以下描述的那样，自左向右沿着l_i扫描一遍。假设沿着某条边e进入了面f。l_i将迟早会在f的另一条边e'对应的一条半边处离开面f。因此接下来，我们将根据双向链接边表中的各个Next()指针，沿着f的边界前行，直到遇到这条边e'。然后，借助于这条半边的Twin()指针，可以找到在双向链接边表中与e'互为兄弟的另一条半边，并由此进入到下一张面中。按照这种方式，可以在正比于f的复杂度的时间内，找到下一张面。当然，我们也可能会碰巧在某个顶点v处穿出并离开面f。果真如此，可以围绕着v扫描一圈，依次访问与之关联的各条边，直到发现与l_i相交的下一张面。借助于双向链接边表结构，可以在正比于v的度数的时间内，完成这样一步计算。图 8-14 所显示的，就是我们对排列进行遍历（traversal）的过程。

这里有两个问题：首先，如何才能找到与 l_i 相交的、最靠左侧的那条边（亦即我们沿着 l_i 在 $\mathcal{A}_{i-1}$ 中经过的第一条边）？其次，每次遇到一张新的面，又如何才能真正地将它一分为二？

前一问题很简单。l_i 与 $\mathcal{A}_{i-1}$ 相交的第一条边，必然是包围框 $\mathcal{B}$(L)的某一条边。只需逐一检查 $\mathcal{B}$(L)的四条边，即可确定究竟应该从何处开始遍历。与该边相关联、位于 $\mathcal{B}$(L)内部的那张面，就是应该被 l_i 一分为二的第一张面。倘若碰巧 l_i 与 $\mathcal{A}_{i-1}$ 相交于 $\mathcal{B}$(L)的某一角落处，则与该角点关联的、位于 $\mathcal{B}$(L)内部的那张面也是唯一确定的，而它正是应该被 l_i 一分为二的第一张面。若直线 l_i 是垂直的，则可以找到 l_i 与 $\mathcal{A}_{i-1}$（中的各直线）之间高度最低的那个交点，然后从该点处开始遍历。由于 $\mathcal{B}$(L)至多为 $\mathcal{A}_{i-1}$ 贡献 2i+2 条边，故对于每一条直线，这一步计算都只需线性时间。

现在，假定要分割面f，而且与f相邻于左侧、同样与l_i相交那张面已经被分割过了。特别地，可以假定进入面f时所跨越的那条边e也已经被分割过了。为分割面f，可采用下列方法（参见 图 8-15）。以直线l_i为界，f将被分割成上、下两半。因此，我们首先要生成两个新的面记录，分别对应于新的这两张面。接下来，在l_i离开面f时所跨越的那条边e'处，也要对其进行分割，并生成一个对应于$l_i \cap e'$

的顶点。这样，我们不仅生成了一个新的顶点记录，而且对于（由e'分割出来的）两条新边，还要分别生成两个半边记录。（若l_i在离开f时跨越的是某个顶点，则这一步就可以省略掉。）此外，对应于新边$l_i \cap f$，也要生成（两条）半边。剩下的工作不过是：对新的面记录、顶点记录以及半边记录中的各个指针正确地进行初始化；对已有的指针进行设置，使之正确地指向相应的顶点记录、半边记录以及面记录；销毁对应于f的面记录、对应于e'的（两个）半边记录。第 2.3 节曾经构造过任意两个子区域划分的叠合，只要参照当时所采用的方法，就可以完成上述任务。而每次分割计算所需的时间，将线性正比于f的复杂度。

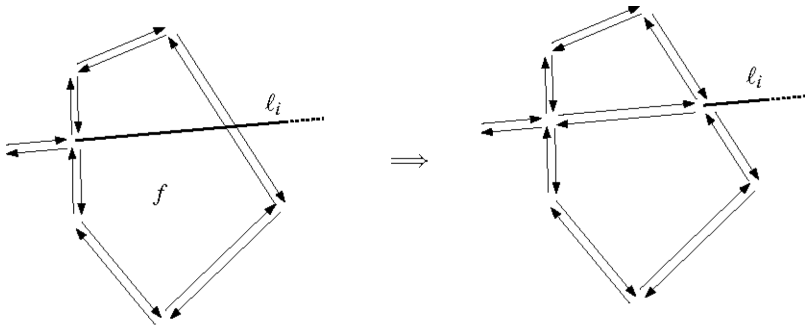


图8-15 面的分割

构造排列的算法，可以总结如下：

```
算法    CONSTRUCTARRANGEMENT(L)
输入：由平面上 n 条直线组成的一个集合 L
输出：一个双向链接边表结构，
          对应于由限制于 B(L)内部的 B(L)、A(L)共同导出的子区域划分
          其中 B(L)为一个包围框，它包含了 A(L)中的所有顶点
1.   构造一个足以包容下 A(L)中所有顶点的包围框 B(L)
2.   对应于由 B(L)导出的子区域划分，构造一个双向链接边表结构
3.   for (i = 1 to n)
4.     do 在B(L)（的边界）上，找到l_i与A_{i-1}[①]的交点所在的那条边e
5.        f ← 与 e 相关联的那张有界面
6.        while (f 不是无界面)   (* 也就是说，f 不是位于 B(L)外部的那张面 *)
7.          do 将 f 一分为二，并将 f 设为下一张（与 l_i）相交的面
```

以上给出了一个简单的递增式算法，用以构造排列。接下来，要对其运行时间做一分析。算法的第 1 步是构造 $\mathcal{B}(L)$，这可以在 $O(n^2)$时间内完成。第 2 步只需常数时间。正如此前已经说明过的，

① 原书误作$\mathcal{A}_i$。——译者

为了找到被 l_i 一分为二的第一张面，只需消耗 $O(n)$时间。我们最后的任务就是要界定出，为了将与 l_i 相交的那些面分割开来，总共需要多少时间。

首先假定 $A(L)$是简单的。这样，为分割一张面 f 并找出与 l_i 相交的下一张面，所需的时间将线性正比于 f 的复杂度。因此，为插入直线 l_i（到 A_{i-1} 中）所需的时间，将线性正比于 A_{i-1} 中与 l_i 相交的各张面的复杂度之和。倘若 $A(L)$不是简单的，则可能会跨越某个顶点 v 以进入下一张面。这种情况下，必须围绕着 v，逐一扫描与之相关联的各条边，以找出需要分割的下一张面——这一过程中所遇到的各边，不见得都属于某张（与 l_i）相交面的边界。然而我们还是注意到：即使是在这种情况下，以我们所访问到的任一条边为边界的面，（尽管本身可能会与 l_i 不相交，但是）其闭包（closure）必然与 l_i 相交。由此，便引出了所谓“带域”（zone）的概念。

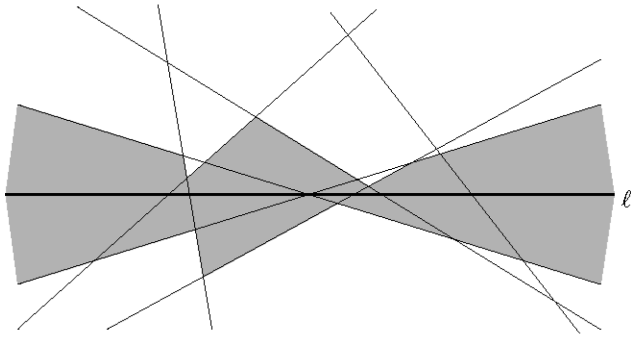


图8-16 在一个直线排列中，一条直线所对应的带域区域

任给平面直线集L，均可导出一个排列$A(L)$。任一直线l在$A(L)$中对应的带域，是由$A(L)$中若干张面组成的一个集合；一张面属于该集合，当且仅当其闭包与l相交。图 8-16 中给出了由 9 张面组成的一个带域的实例。所谓带域的复杂度，就是其中所有面的复杂度——边数、顶点数——之和。从 图 8-16 可以看出，在计算带域的复杂度时，有些顶点被统计了一次，另一些则被统计了两次、三次，甚至四次。为插入直线l_i所需的时间，将线性正比于l_i在$A(\{l_1, \ldots, l_i\})$中所对应带域的复杂度。而下面的带域定理（zone Theorem）则指出，这一数量实际上是线性的。

〖定理 8.5（带域定理）〗

在由平面上任意 m 条直线构成的排列中，任一直线所对应带域的复杂度都是 $O(m)$。

〖证明〗

设 L 为由平面上 m 条直线构成的一个集合，l 为另外的一条直线。不失一般性地，假定 l 与 x-坐标重合——通过对坐标系的旋转，这一条件必然可以满足。我们还假定 L 中的任何线段都不是水平的——在本证明的最后，这一假设条件将被取消。

在 $A(L)$中，每一条边都同时属于两张面的边界。相对于其右侧的那张面，这条边称作左侧包围边（left bounding edge）；而相对于其左侧的那张面，我们称之为右侧包围边（right

bounding edge）。我们将证明：沿着 l 所对应的带域，各张面的左侧包围边的总数至多为 5m。根据对称性，右侧包围边的总数也不会超过 5m，于是本定理得证。

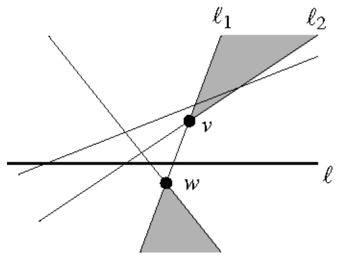


图8-17 带域中各张面的左侧包围边的总共不超过5m条

我们通过对 m 的归纳来证明。对于最基本的情况（即 m = 1 时），显然是成立的。现在假设 m > 1。在 L 内的各条直线中，令 l_1 为与 l 相交于最右侧的那一条。首先假设，这条直线是唯一确定的。根据归纳假设，l 在 $A(L\backslash\{l_1\})$中所对应的带域，最多拥有 5(m-1)条左侧包围边。当插入直线 l_1 后，会增加一定数量的左侧包围边，不过，这种增长的来源不外乎两种：要么是 l_1 所贡献的新的左侧包围边，或者是原先已有的左侧包围边被 l_1 一分为二。在 l_1 与 L 中各直线的交点中，找出位于 l 上方的第一个（即最低的那个）——令这个交点为 v；找出位于 l 下方的第一个（即最高的那个）——令这个交点为 w。以 v 和 w 为端点的那条线段，就是 l_1 所贡献的一条新的左侧包围边。此外，在点 v 和 w 处，l_1 还分别将原先的一条左侧包围边一分为二。这样加起来，左侧包围边就增加了 3 条。如果 v 或者 w 不存在，增加的数目甚至还没有这么多。我们声称，除此之外，不会有任何其它形式的增加。

考察 l_1 位于 v 上方的部分。将与 l_1 相交于 v 点的另一条线段记作 l_2。由 l_1 和 l_2 围成的、位于 v 上方的那个（连通的）子区域，并不属于 l 所对应的带域。因为在 v 点处，l_2 自左向右地穿越 l_1，所以这个子区域必然位于 l_1 的右侧。既然如此，l_1 位于 v 上方的部分就不可能为该带域贡献任何左侧包围边。不仅如此，对于任何一条原来属于该带域的左侧包围边 e，只要 e 与 l_1 相交，而且其交点位于 v 的上方，那么沿着 e、落在 l_1 右侧的那一段必然不再属于该带域。因此，尽管存在这些交点，它们都不会使左侧包围边的数目有所增加。

同理可证，l_1 位于 w 下方的部分，也不可能使 l 所对应带域中的左侧包围边数目增加。因此，正如我们所声称的那样，（左侧包围边）增加的数目充其量不会超过 3。在这种情况下，左侧包围边的总条数不会超过 5(m-1) + 3 < 5m。

到目前为止，我们一直假定，穿过 l 上最右端那个交点的直线只有 l_1 这唯一的一条。要是这一条件不成立（即有多条直线同时穿过最右端的这个交点），我们实际上可以任取其中一条做为 l_1。按照与上面相同的推理方法可以证明，左侧包围边增加的数量至多不过是 5。（如果有两条以上的直线同时穿过这个交点，则增加量不会超过 4；而在恰好只有两条直线同时穿过时，增加量才最多可能达到 5。）于是，左侧包围边的总数至多为 5(m-1) + 5 = 5m。

最后，我们来消除“L 中不含水平直线”这一假定条件。对于不与 l 重合的任何一条直线，

任何足够轻微的旋转，都会导致 l 在 $\mathcal{A}$(L)中所对应带域的复杂度增加。因为待证明的只是一个上界，所以完全可以放心大胆地假设“这类直线根本就不存在”。即使 L 中某条直线 l_i 与 l 碰巧重合，根据上述证明的结论，l 在 $\mathcal{A}(L\setminus\{l_i\})$中所对应的带域，只多含有 10m - 10 条边。此时再引入 l_i，这一数量最多只会增加 4m - 2——来自于 l_i、对应于 l_i 上方某张面的边，至多有 m 条；来自于 l_i、对应于 l_i 下方某张面的边，也至多有 m 条；另外，至多有 m-1 条边被一分为二，这些边中的每一条，或者被记入到左侧包围边的总数中，或者被记入到右侧包围边的总数中。于是，带域定理得证。 ❑

至此，才可以确定我们用来构造排列的递增式算法的运行时间上界。我们已经知道：为了插入 l_i 所需的时间，线性正比于 l_i 在 $\mathcal{A}(\{l_1, \ldots, l_{i-1}\})$中所对应带域的复杂度。根据带域定理，也就是 $O(i)$，因此，插入所有直线总共所需的时间为：

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

算法的第 1 到 2 步，加起来需要 $O(n^2)$时间，故该算法的总体运行时间为 $O(n^2)$。只要 $\mathcal{A}$(L)是简单的，$\mathcal{A}$(L)的复杂度就必然是$\Theta(n^2)$，因此，我们的这一算法已经是最优的了。

8.4 层阶与偏差

现在可以回到最初的差异值问题了。我们已经通过对偶变换，将由n个采样点组成的集合S，转化为由n条直线组成的集合S^*；我们的任务，也转化为“对于$\mathcal{A}(S^*)$中的每一个顶点，计算出位于其上方、下方以及正好经过它的直线各有多少条”。对于任何一个顶点，这三个数之和必然等于n，故计算出其中任意两个即可。只要构造出$\mathcal{A}(S^*)$所对应的双向链接边表结构，也就知道了各有多少条直线经过每个顶点。如 图 8-18 所示，对于直线排列中的每个点，我们将严格位于其上方的直线条数，定义为该点的层阶（level）。下面就来看看，应该如何计算出$\mathcal{A}(S^*)$中各顶点的层阶。

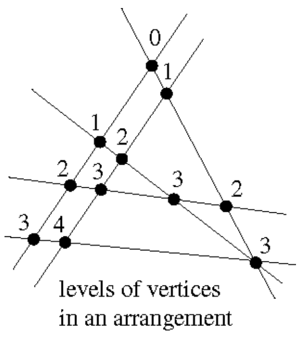


图8-18 排列中各顶点的层阶

为了得到$\mathcal{A}(S^*)$中各顶点的层阶，对每条直线 $l\in S^*$，我们都需要进行以下计算。首先，要计算出 l 上最靠左端那个顶点的层阶。为此，只要逐一检查其余的各条直线，分别确认它们是否严格地位于这一顶点的上方。因此，这一步需要 $O(n)$时间。然后，借助于双向链接边表结构，可以沿着 1 自左向右依次访问 1 上其余的各个顶点。在这个遍历的过程中，层阶是很容易维护和更新的——只有在顶点处，层阶才可能会有所变化。每次遇到一个顶点，只要检查与之相关联的所有边，就可以知道层阶的变化量具体为多少。

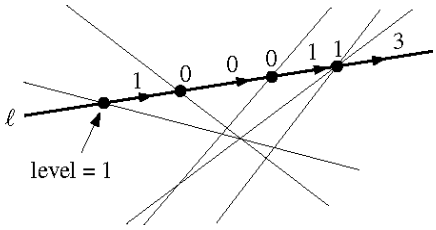


图8-19 在沿着一条直线行进的过程中，动态维护层阶

以 图 8-19 为例，l上最靠左端那个顶点的层阶为 1。从该顶点出发，沿着与之相关联的那条边向右开始的一段上，所有点的层阶也都是 1。在（沿l自左向右的）第二个顶点处，有一条直线自上而下地跨越了l；于是，层阶减一，变成 0。此外，根据定义，一个点的层阶指的是“严格位于该点上方的直线条数”，因此，第二个顶点本身的层阶也应是 0。在第三个顶点处，又有另一条直线自下而上地跨越了l。因此，在经过这个顶点之后，层阶将增长到 1；特别地，这个顶点本身的层阶仍然为 0。后面的情况依此类推。请注意，我们并不需要担心会有垂线出现——因为，这个（直线的）集合是通过对偶变换，从一组点转化过来的。这样，就可以在$O(n)$时间，计算出l上所有顶点各自的层阶。而整个$\mathcal{A}(S^{*})$中所有顶点的层阶，将可以在$O(n^2)$时间内计算出来。

这样，我们就知道了，在$\mathcal{A}(S^{*})$中各顶点的上方、下方各有多少条直线，而且有多少条直线正好穿过各顶点。现在，对于由S中任意两点确定的一条直线，我们都可以计算出，以该直线为边界的半平面的离散测度等于多少。也就是说，所有这类离散测度都可以在$O(n^2)$时间内计算出来。至此，〖定理 8.2〗的证明才最终得以完成。

8.5 注释及评论

本章介绍了若干重要的非算法性概念——几何对偶（geometric duality）以及排列。在求解几何问题的时候，对偶变换可以使我们从另一个角度来进行思考；而对于计算几何学家们来说，对偶变换已经成为了一个标准的工具。第 8.2 节所介绍的那种对偶变换，对垂线没有定义。通常，既可以将垂线作为一种特殊情况专门加以处理，也可以通过对设置描述的轻微扰动来加以解决。事实上，能够处理垂线的对偶变换有好几种，不过，它们都同时存在某些缺点——有关这一方面的详细介绍，

请参见Edelsbrunner的专著 [158]。高维空间中的点集，同样可以进行对偶变换。对于任一点$p = (p_1, p_2, \dots, p_d)$，其对偶$p^*$为一张超平面：$x_d = p_1x_1 + p_2x_2 + \cdots + p_{d-1}x_{d-1} - p_d$。反过来，对于任一超平面：$x_d = a_1x_1 + a_2x_2 + \cdots + a_{d-1}x_{d-1} - a_d$，其对偶为一个点$(a_1, a_2, \dots, a_{d-1}, -a_d)$。在这种变换下，关联性以及次序都是保持的。

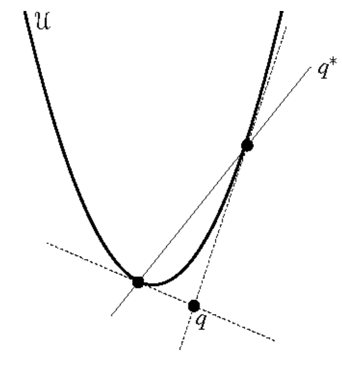


图8-20 借助抛物线U确定点q的对偶直线q*

你应该记得，借助抛物线$y = x^2/2$，可给出这种对偶变换的几何解释，并由此构造出任何点的对偶（直线）。有趣的是，根本不用测量距离，即可构造出任一点q的对偶。如 图 8-20 所示，从q出发画出U的两条切线。如此，点q也可理解为这两条切线的交；反过来，q的对偶直线必然会经过这两条切线的对偶点——而这两个点就是这两条切线与U的切点。对于位于U上方的点，也无须测量距离，即可构造出其对偶。此处略去具体的构造方法。（提示：你必须能够完成这样的操作：给定一条直线以及该直线外的一个点，画出经过这个点、与这条直线平行的另一条直线。）

还有另一种几何对偶，也在计算几何中得到了成功的应用，这就是所谓的反演（inversion）。通过这种几何对偶，可以将平面上“点位于圆内”的关系，转化为三维空间中“点位于平面下方”的关系。更详细地说，任何一个点 $p := (p_x, p_y)$都被投影到三维空间中的一个单位抛物面 $z = x^2 + y^2$ 上——也就是说：

$$p^o \ := \ (p_x, p_y, p_x^2 + p_y^2)$$

平面上的任何一个圆$C := (x-a)^2 + (y-b)^2 = r^2$，都可以按照下列方法，变换为三维空间中的某张平面：将该圆（上的各点）投影到单位抛物面上，然后取投影所在的那张平面[①]。具体地讲，就是

$$C^o \ := \ (z = a(x-a) + b(y-b) + r^2)$$

这样，点 p 落在 C 的内部，当且仅当 p^o 位于 C^o 的下方。这个变换可以推广至高维空间——这种情况下，d-维空间中的一个超球（hypersphere），就相当于(d+1)-维空间中的一张超平面。

① 读者可以自己证明：同一圆上的任何一点，垂直投影到这个抛物面上所得的影像，都是共面的。——译者

在计算几何学与组合几何学（combinatorial geometry）中，都对排列做了充分的研究。排列的定义范围并不只限于平面。任意给定一组平面，也就相应地导出了一个三维的排列；而若是一组超平面，则会相应地导出一个更高维的排列。在Edelsbrunner的专著 [158]中，对 1987 年之前有关排列的研究做了出色的剖析。其中还提供了一些参考文献索引，介绍了组合几何学——而不是计算几何学——的一些早期教材。如果希望获得这方面最新发展的综述，读者可以参见Halperin专著中的相关一章 [206]。以下仅选取并罗列出若干关于平面及更高维空间中的排列的研究成果。

由d-维空间中n张超平面构成的排列，在最坏情况下的复杂度为$\Theta(n^d)$。任一简单排列——也就是其中任何d张超平面都相交于一点，而且其中任何d+1 张超平面都不会相交于一点——都会达到这个复杂度。在排列的构造方面，Edelsbrunner等人 [165]首次给出了一个最优的算法。他们这个递增式算法的最优性，取决于带域定理的高维空间中的一个推广——这个更具一般性的定理指出：任意给定由d-维空间中n张超平面所构成的一个排列，任何一张超平面所对应带域的复杂度为$O(n^{d-1})$。这则定理的证明，也是Edelsbrunner等人给出的 [168]。

排列中的层阶概念，同样可以推广到高维空间——这方面的介绍也请参阅Edelsbrunner的专著[158]。给定由任意n张超平面导出的一个排列$A(H)$，其中的所谓k-层阶（k-level），就是由最多有k-1张超平面严格地位于其上方、最多有n-k张超平面严格地位于其下方的那些点所构成的集合。k-层阶的最大复杂度是多少？至今还没有得到一个紧的上界——即使是在平面上，亦是如此。从对偶的角度来看，这一问题与这样一个问题紧密相关：（给定k，）对任意的一组共n（>k）个点，通过一张超平面，可以将其中的某k个点与其余的n-k个点分隔开来，构成原点集的一个子集，这样的子集称为一个“k-子集”（k-set）；那么，k-子集共有多少个呢？同样地，对于由任意n个点构成的点集，其中k-子集的数目也是未知的。就平面的情况而言，Erdos等人 [174]在 1973 年证明：无论是k-子集还是k-层阶，下界都是$\Omega(n\log(k+1))$，上界都是$O(n\sqrt{k})$。除了Pach等人 [313]所做的微小改进（他们将上界改进至$O(n\sqrt{k} / \log^*(k+1))$）之外，这个问题可以说一直悬而未决。直到 1997 年，Dey才成功地将上界改进至$O(nk^{1/3})$——这是至今为止最好的结果。

对于平面上任意n个点构成的集合，通过一条直线，都可以将其中的不超过k个点（与其余的n-k[①]个点分隔开来）构成一个子集，称作一个“(≤k)-子集”（(≤k)-Set）。如果k是固定的，那么(≤k)-子集的数目又是多少呢？与k-子集的情况不同，(≤k)-子集最大数目的紧上界已经为我们所知。就平面情况而言，其最大数目为$\Theta(nk)$；在一般的d-维空间中，这个紧上界为$\Theta(n^{\lfloor d/2 \rfloor}k^{\lceil d/2 \rceil})$——这个结果是由Clarkson和Shor[133]证明的。这个上界，同样适用于排列中的(≤k)-层阶（(≤k)-level）。

第 7 章的“注释及评论”一节，揭示了Voronoi图与高一维空间中的凸多面体之间的联系——任意平面点集对应的Voronoi图，与三维空间中一组半空间的公共交集的边界的投影完全一致。实际上，

[①] 原书此处叙述不够严谨。按照(≤k)-子集的定义，分割出来的这个子集实际所含的点数为 $t \le k$，故剩下的点应该有 $n-t \ge n-k$ 个。——译者

这个边界就是界定这些半空间的那些平面所构成的排列中的 0-层阶。这种联系，可以推广到k-阶Voronoi图与排列中的k-层阶之间——在由平面构成的同一排列中，将k-层阶向下投影（到xy-平面），就得到了这些点所对应的k-阶Voronoi图。

除了直线和超平面，对其它类型的对象也可以定义排列。例如，平面上的任意一组线段，也构成一个排列。对于这类排列，即使是证明单张面所能达到的最高复杂度的界，也绝非一件易事。在这种情况下，面不见得是凸的，因此同一条线段可以在边界上多次出现。实际上，即使是单张面的最高复杂度，也可能是超线性的——在最坏情况下，它是$\Theta(n\alpha(n))$，其中的$\alpha(n)$是所谓的Ackermann逆函数（functional inverse of Ackermann's function），该函数的增长速度极其缓慢。借助于Davenport-Schinzel序列（Davenport-Schinzel sequence），可以证明这个上界；如果读者对此感兴趣，可以参阅Sharir和Agarwal合写的专著 [353]。

对诸如排列、排列中的各个单元以及包络（envelope）等组合结构的研究目的，在于运动规划。很多运动规划方面的问题，都可以借助排列及其结构来描述 [201][207][208][231][342][343]。

对于我们来说，研究排列的最初动机，来自于计算机图形学以及随机采样质量等方面的问题。差异值的方法，由Shirley[358]首先引入到计算图形学中；此后，Dobkin和Mitchell[150]、Dobkin和Eppstein[149]、Chazelle[96]以及Berg[50]等人在算法方面又有所发展。

8.6 习题

习题 8.1 试证明：正如〖观察结论 8.3〗所声称的，本章所介绍的对偶变换，的确能够保持关联性并保持次序。

习题 8.2 正如第 8.2 节所指出的，一条线段的对偶，是一个左右式的双楔形。

a. 给定顶点分别为p、q和r的一个三角形，若将该三角形的所有内点构成一个集合，该集合的对偶是什么？

b. 原平面上何种类型的对象，其对偶为一个上下式的双楔形？

习题 8.3 试利用欧拉公式证明：对于含有 $n(n-1)/2$ 个顶点和 n^2 条边的排列来说，其中所含面的数目最多不会超过 $n^2/2 + n/2 + 1$。

习题 8.4 给定由平面上 n 条直线组成的一个集合 L。试给出一个 $O(n\log n)$时间的算法，构造出一个与坐标轴平行的矩形，将$A(L)$中的所有顶点都包含在其内部。

习题 8.5 给定由平面上 n 个点组成的一个集合 S。本章曾经给出了一个算法，可以对 S 中每一对点所确定的一条直线 l，计算出 S 中有多少个点严格地位于 l 的上方。这个算法首先要对这个问题做对偶变换。现在，试将经过对偶变换后所得的问题，反过来转换回到原平面之中，并且这对给定的这一问题，给出一个 $O(n^2)$的求解算法。（在做过这

道习题之后，相信你会认识到对偶变换的威力。）

习题 8.6 给定由平面上 n 个点组成的一个集合 S，以及由平面上 m 条直线组成的集合 L。现在，我们希望判断出来，S 中是否有某个点正好落在 L 中的某条直线上。该问题的对偶问题是什么？

习题 8.7 在平面上给定 n 个红色的点和 n 个蓝色的点，它们分别组成集合 R 和 B。如果存在某条直线 l，使得 R 中的各点都位于 l 的一侧，而 B 中的各点都位于 l 的另一侧，我们就称 l 为一条分隔线（separator）。试给出一个随机算法（randomized algorithm），对于任意给定的 R 和 B，在 $O(n)$的期望时间内，判断它们之间是否存在一条分隔线。

习题 8.8 第 8.2 节介绍的对偶变换中存在减号。现在，我们将其中的这些减号替换为加号——也就是说，点(p_x, p_y)的对偶直线为$y = p_x x + p_y$；反过来，直线$y = mx + b$的对偶点为(m, b)。这样的一个对偶变换，是否仍然会保持关联性和次序？

习题 8.9 给定由平面上 n 个点组成的一个集合 P。任选其中的一个点 $p \in P$。试给出一个随机算法，在 $O(n)$的期望时间内，判断出 p 是否为 P 的凸包上的一个顶点。

习题 8.10 给定由平面上 n 条非垂直线组成的一个集合 L。假设在排列 $A(L)$中，所有顶点的层阶都等于 0。对于这样一个排列，你可以做出什么断言？然后，再假设 L 中的直线可以是垂直的。对于这样的排列，你又可以做出什么断言？

习题 8.11 给定由平面上若干条直线组成的一个集合 L，令 f 为排列 $A(L)$中坐标原点所在的那张面。f 内部各点的对偶直线，合起来是什么样子？试对其做一描述。另外，f 各个顶点的对偶直线又是什么样子的？你必须对 f 的顶点分几种情况讨论：由同时位于原点上方的两条直线相交而得的顶点；由同时位于原点下方的两条直线相交而得的顶点；以及由分别位于原点上方、下方的两条直线相交而得的顶点。

习题 8.12 在构造某一直线集 L 所对应的排列时，每引入其中的一条直线，都要对它自左向右地遍历一次。而为了计算出差异值，我们需要知道排列中各个顶点的层阶。为了计算出这些层阶，也需要自左向右地遍历 L 中的每一条直线。既然如此，能否将这两次遍历合二为一呢？——也就是说，在将一条直线添加到排列之中的过程中，我们能否及时地计算出各交点的层阶呢？

习题 8.13 给定由平面上任意 n 条直线组成的集合 L，试给出一个 $O(n\log n)$的算法，计算出排列 $A(L)$中所有顶点的最大层阶。

习题 8.14 给定由平面上任意 n 个点组成的一个集合 S。试给出一个运行时间为 $O(n^2)$的算法，找出经过 S 中各点数目最多的那条直线。

习题 8.15 给定由平面上任意 n 条线段组成的一个集合 S。我们希望通过预处理，将 S 转化为某种数据结构，以回答这样的查询：给定一条查询直线 l，l 与 S 中的多少条线段相交？

a. 试在对偶平面中重新描述这一问题。

b. 试给出解决这一问题的一种数据结构，该结构使用 $O(n^2)$的期望存储空间，其对应的期望查询时间为 $O(\mathbf{log}n)$。

c. 试描述，如何才能在 $O(n^2\mathbf{log}n)$期望时间内构造出这一结构。

习题 8.16 给定由平面上任意n条线段组成的一个集合S。如 图 8-21 所示，如果一条直线l与S中的所有线段都相交，l就被称为是S的一条“截线”（transversal），或者S的一个“穿刺”（stabber）。

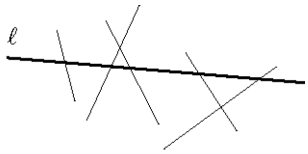


图8-21 直线集的截线

a. 试给出一个算法，对任何集合 S，判断出 S 是否存在一个穿刺。

b. 现在，假设所有的线段都是垂直的。试给出一个随机算法，对任何集合 S，都能在 $O(n)$的期望时间内，判断出 S 是否存在一个穿刺。

9

Delaunay三角剖分：高度插值

前面的章节曾经谈到过地图，只不过，其涉及的范围仅限于地球表面局部的某块（如 图 9-1 所示），并做了一个隐含的假设——地形的高低变化可以忽略。对于荷兰这样的国家，这一假设或许不无道理；但要是换成瑞士，这个假设就不能成立了。本章就来解决这个问题。

借助一种称为地形（terrain）的结构，也可以表示地表的一块区域。所谓地形，是定义于三维空间之中的二维表面，且具有如下特殊性质：如果它与某条垂线相交，那么最多相交于一点。换而言之，它也就是某个函数$f: A \subset \mathbb{R}^2 \mapsto \mathbb{R}$对应的图，对于该地形的定义域（domain）A中的每个点，这

个函数都指定了一个高度。（地球是圆的，因此如果考察的是整个地球，这就不是一种适宜的建模方式。不过就局部范围而言，地形的确是一种相当好的建模方式。）为了使地形可视化，我们既可以通过透视的方式将这个图直接画出来（如 图 9-2 所示），也可以象通常的地形图（topographic map）那样，绘制出等高线（contour line）——即由高度相等的点联接而成的线条。

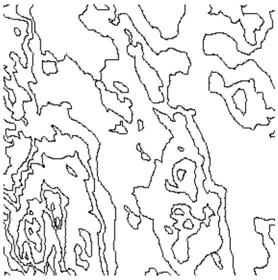


图9-1 局部地图

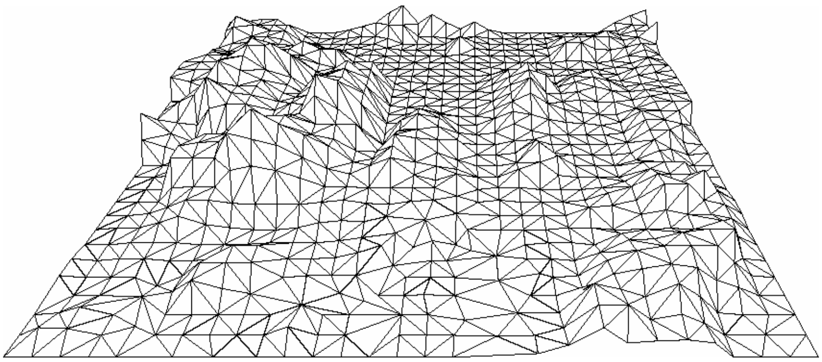


图9-2 地形的透视显示

当然，不可能知道地球上每一处的高度；我们所掌握的，只是实际测量过的那些位置的高度。这就是说，在论及地形的时候，我们对函数f取值分布的了解，只限于某个采样子集$P \subset A$内的有限个样本点。为了得到其定义域内其它位置的高度，不得不借助那些已知的高度，以某种方式给出近似值。一种朴素的做法，就是对于任何点$p \in A$，都找到与之最靠近的一个采样点，然后将p 的高度设置为这个样本的高度。然而，如 图 9-3 所示，如此生成的地形并不连续，外观很不自然。

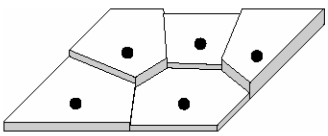


图9-3 不连续的地形

鉴于这一问题，可采用如下方法近似地形。首先，对P做三角剖分——由此生成平面的一个子区域划分（subdividsion），其中每张有界的面都是三角形，而其中的顶点就是来自于P的各点。（这里假设，所选采样点允许我们通过构造三角面片，来覆盖该地形的整个定义域。）然后，将每个采样点提升

至其对应的高度——这样，原来三角剖分中的每个三角形，都将转化为三维空间中的一个三角形。如 图 9-4 所示。如此可得一个所谓的多面式地形（polyhedral terrain），它可看作是一个分段线性的（piecewise linear）连续函数的图象。利用这种多面式地形，可以近似实际的地形。

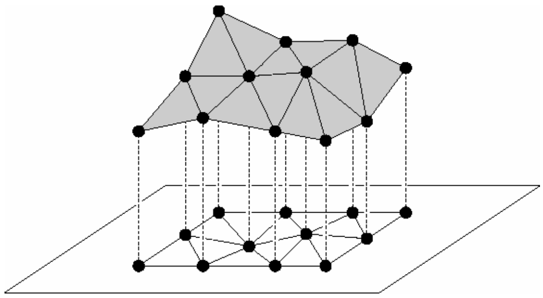


图9-4 由一组采样点，得到一个多面式地形

现在的问题是：给定一组采样点P，应如何对它们做三角剖分？一般而言，有多种方法。然而针对这里的特殊要求（即对地形进行近似），哪一种三角剖分才最为适合呢？没有确定的答案。我们并不知道实际的地形，只知道在若干采样点处的高度值。我们没有其它的任何信息，但无论是何种三角剖分，所有采样点处的高度值却总是正确的。由此看来，无论对P怎样做三角剖分，效果似乎都差不多。然而，就其外观的自然性而言，某些三角剖分的确会更好。图 9-5 给出了同一个采样点集的两个三角剖分，让我们来作一对比。根据各采样点的高度值，我们可以感觉出来，它们都是沿着一道山脊分布的。三角剖分(a)与我们的这种直觉是吻合的。然而在(b)中，因为有一条边被“翻转”了，从而出现了一道山谷，横贯跨越在该山脊之上。从直觉上看，这是错误的。

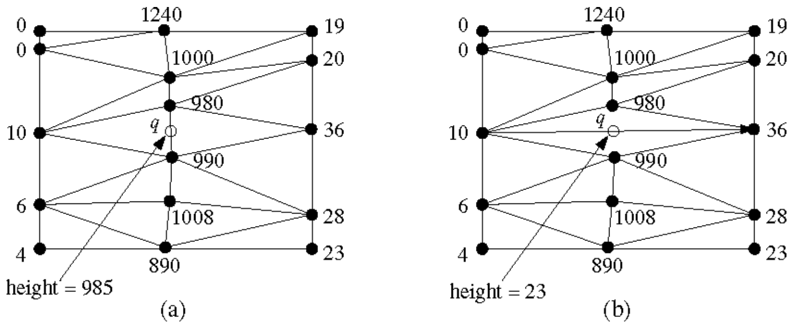


图9-5 哪怕只翻转一条边，也可能会有天壤之别

那么，能否将这一直觉归纳为某种准则，根据这一准则，我们可以说三角剖分(a)的确要优于(b)呢？

三角剖分(b)的问题在于，点 q 处的高度是根据距离相对更远的两个点得出的。如果 q 恰好落在两个狭长的三角形之间的公共边上，就会出现这种问题。也就是说，问题的根源在于这两个三角形的外形过于狭长。如此看来，要是三角剖分中含有过小的夹角，就是不好的。于是，我们可以将各三角剖分中所含的最小夹角作为一个指标，来衡量它们的优劣。如果两个三角剖分中所含的最小夹角相等，我们就比较其中的次小角度，……，依此类推。对于任一给定的点集 P，可能的三角剖分

的总数必是有限的。因此，必然存在一个使最小夹角达到最大的三角剖分，（根据这里所强调的准则，）它就是最优的三角剖分。我们的目标，就是找出这个三角剖分。

9.1 平面点集的三角剖分

任取平面上的一个点集 $P := \{p_1, \ldots, p_n\}$。为了能够对 P 的三角剖分做一形式化定义，我们首先要引入极大平面子区域划分（maximal planar subdivision）这一概念。所谓的极大平面子区域划分，是一种特殊的子区域划分——如果试图在其中任意两个（没有直接相联的）顶点之间引入一条新的边，都将破坏其平面性（planarity）。换而言之，任何不属于 S 的边，必然与 S 中已经存在的某条边相交。这样，我们就可以将 P 的一个三角剖分，定义为“以 P 为顶点集的一个极大平面子区域划分”。

按照这一定义，三角剖分的存在性是显而易见的。问题在于，如此定义的三角剖分，一定是由三角形组成的吗？是的，除了唯一的那张无界面外，其它的所有面都是三角形——任何有界面都是一个多边形，而根据第 3 章的结论，任何多边形都可以被三角化。那么，无界的那张面又会如何呢？不难看出，无论是在哪一个三角剖分$\mathcal{T}$中，沿着P的凸包边界，联接任何两个相邻点的线段，都必然是$\mathcal{T}$中的一条边。由此可以得知，$\mathcal{T}$中全部有界面的并，必然恰为P的凸包。（这也意味着，就这里的应用问题而言，我们必须保证地形定义域（比方说一个矩形区域）的角点必须被采样到——这样才能保证三角剖分内的三角形合起来能够覆盖地形的整个定义域。）

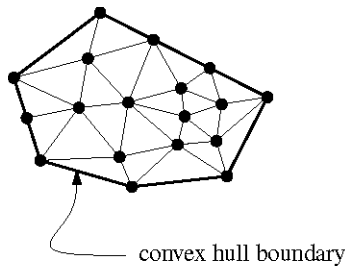


图9-6 同一点集的三角剖分含同样数目的三角形，具体数目取决于点集的规模及其凸包的规模

如图 9-6 所示，对于同一点集P，任何三角剖分中所包含的三角形数目都是相等的；其中包含的边数也是相等的。具体数目等于多少，取决于落在P的凸包边界上的点数。（如果某个点落在凸包边界上某条边的内部 [①]，我们在这里也把它统计在内。因此，落在凸包边界上的点数，不见得正好等于凸包的顶点数目。）下面这则定理，对此做了准确的表述：

[①] 比如，互异的三个共线点同时落在凸包的边界上。——译者

〖定理 9.1〗

设 P 为由平面上不全部共线的任意 n 个点组成的一个集合，落在 P 的凸包边界上点的个数记作 k。则 P 的任何一个三角剖分必然由 2n - 2 – k 个三角形组成，而且共有 3n - 3 – k 条边。

〖证明〗

任取 P 的一个三角剖分$\mathcal{T}$，将$\mathcal{T}$中三角形的数目记作 m。请注意，如果将该三角剖分中所含面的数目记作 n_f，则有 $n_f = m + 1$。其中，每张三角面都是由三条边构成的，而无界的那张面则是由 k 条边围成的。此外，每一条边都恰与两张面相关联。因此，$\mathcal{T}$中所含边的总数应为 $n_e := (3m + k) / 2$。而欧拉公式告诉我们：

$$n - n_e + n_f = 2$$

将 n_e 与 n_f 的值代入这个公式，就得到了：

$$m = 2n - 2 - k$$

由此进而可以得到，$n_e = 3n - 3 - k$。 ❑

任取 P 的一个三角剖分$\mathcal{T}$，假设其中含有 m 个三角形。考察$\mathcal{T}$中各三角形的总共 3m 个角度，并将它们按照升序排序。设排序结果为$(\alpha_1, \alpha_2, \ldots, \alpha_{3m})$，也就是说，对任何 $i < j$，都有$\alpha_i \le \alpha_j$。我们将 $A(\mathcal{T}) := (\alpha_1, \alpha_2, \ldots, \alpha_{3m})$称为$\mathcal{T}$的角度向量（angle-vector）。取同一集合 P 的另一个三角剖分$\mathcal{T}'$，令 $A(\mathcal{T}') := (\alpha'_1, \alpha'_2, \ldots, \alpha'_{3m})$为与之对应的角度向量。按照字典序，如果 $A(\mathcal{T})$大于 $A(\mathcal{T}')$，我们就说“$\mathcal{T}$的角度向量大于$\mathcal{T}'$的角度向量”。具体地，也就是存在一个下标 i，$1 \le i \le 3m$，满足

对一切 $j < i$，都有$\alpha_j = \alpha'_j$；但$\alpha_i > \alpha'_i$

这种情况，记作 $A(\mathcal{T}) > A(\mathcal{T}')$。如果对 P 的任一三角剖分$\mathcal{T}'$，都有 $A(\mathcal{T}) \ge A(\mathcal{T}')$，就称三角剖分$\mathcal{T}$为角度最优的（angle-optimal）。正如本章的引言中所提到的，若我们的目的是要根据一组采样点构造出一个多面式地形，则角度最优的三角剖分的确是再好不过的——正因为此，我们才会对这种三角剖分感兴趣。

接下来要讨论的问题是，什么样的三角剖分才是角度最优的。为此，需要用到下面这则定理，人们经常称之为Thales定理（Thales's Theorem，如 图 9-7 所示）。任意给定三个点p、q和r，它们在q处会定义出两个角度，其中较小的那个记作$\measuredangle pqr$。

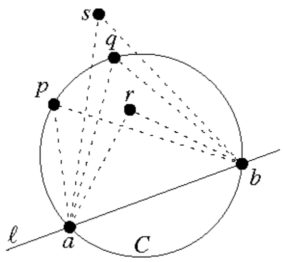


图9-7　通过其相对于弦ab的张角，可以判别园内、圆上以及圆外的点

〖定理 9.2〗

设 C 为一个圆，直线 l 与 C 相交于点 a 和 b；另外，在 l 的同一侧有四个点 p、q、r 和 s。假设 p 和 q 恰好落在 C 上，r 和 s 分别位于 C 的内部和外部。则必有：

$\measuredangle arb > \measuredangle apb = \measuredangle aqb < \measuredangle asb$

现在，针对 P 的任一 $\mathcal{T}$，我们来考察其中的某一条边 e = $\overline{p_ip_j}$。如果边 e 不属于 $\mathcal{T}$ 中那张无界面的边界，它必然会同时与两个三角形 $p_ip_jp_k$ 和 $p_ip_jp_l$ 相关联。如果这两个三角形合起来构成一个凸四边形，那么只要将$\overline{p_ip_j}$从 $\mathcal{T}$ 中删去，代之以$\overline{p_kp_l}$，我们就可以得到另一个三角剖分 $\mathcal{T}'$。这一操作称作边翻转（edge flip）。

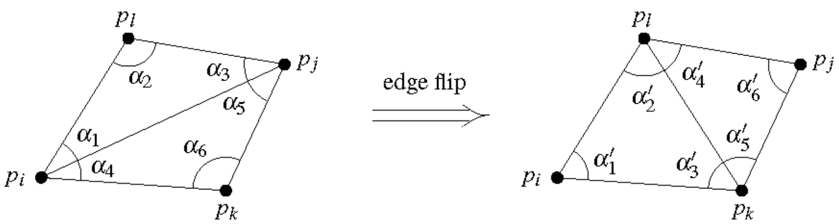


图9-8　边翻转操作

对比$\mathcal{T}$与$\mathcal{T}'$的角度向量，共有六处不同——A($\mathcal{T}$)中的六个角度{α_1、…、α_6}，在A($\mathcal{T}'$)中被换成了{α'_1、…、α'_6}。二者的差别如 图 9-8 所示。如果

$$\min_{1\le i\le 6}\alpha_i < \min_{1\le i\le 6}\alpha'_i$$

我们就将 e = $\overline{p_ip_j}$称作一条非法边（illegal edge）。换而言之，只要在对某条边进行边翻转操作之后，我们能够使局部的（即对应的六个角度中的）最小角增大，它就必然是一条非法边。由非法边的定义，可以立即得出如下观察结论。

〖观察结论 9.3〗

设 e 为三角剖分 $\mathcal{T}$ 中的一条非法边。在 $\mathcal{T}$ 中对 e 进行边翻转操作之后，设新的三角剖分为 $\mathcal{T}'$。则必有 $A(\mathcal{T}') > A(\mathcal{T})$。

实际上，对于给定的任何一条边，无须计算出角度$\alpha_1, \ldots, \alpha_6, \alpha'_1, \ldots, \alpha'_6$的具体数值，即可判断它是否为非法边。相反，可以利用下面这则引理所给出的一个简单的判断准则。如 图 9-9 所示，根据 Thales定理，可以立即证明这一准则的正确性。

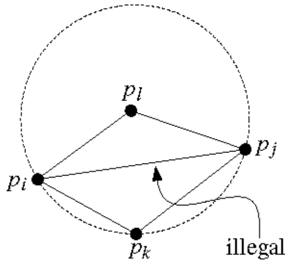


图9-9 根据Thales定理找出非法边

〖引理 9.4〗

设三角形$p_ip_jp_k$和$p_ip_jp_l$之间的公共边为$\overline{p_ip_j}$，令C为由点p_i、p_j和p_k确定的圆。$\overline{p_ip_j}$是一条非法边，当且仅当点p_l落在C的内部。而且，只要点p_i、p_j、p_k和p_l构成一个凸的四边形[①]，并且不共圆，则在$\overline{p_ip_j}$和$\overline{p_kp_l}$二者当中有且仅有一条非法边。

我们也注意到，这条准则对 p_k 和 p_l 来说是对称的——p_l 落在点 p_i、p_j 和 p_k 所确定的圆内，当且仅当 p_k 落在点 p_i、p_j 和 p_l 所确定的圆内。若碰巧这四个点共圆，则$\overline{p_ip_j}$和$\overline{p_kp_l}$都是非法边。请注意：若有两个三角形共有一条非法边，则它们合起来必然构成一个凸的四边形。因此，无论何时，一旦发现一条非法边，都可以立即对它实施边翻转操作。

不含任何非法边的三角剖分，称作合法三角剖分（legal triangulation）。由上面的观察结论可知，角度最优的三角剖分必然也是合法三角剖分。任给一个初始的三角剖分，我们都可以很容易地构造出一个合法三角剖分。为此，只需反复地进行边翻转操作，直到所有的边都已合法。

算法 LEGALTRIANGULATION($\mathcal{T}$)

① 亦即，这四个点的凸包是一个四边形。——译者

输入：点集 P 的任一三角剖分

输出：P 的一个合法三角剖分

1. **while** ($\mathcal{T}$ 中至少还有一条非法边$\overline{p_ip_j}$)
2. **do**(* 翻转边$\overline{p_ip_j}$ *)
3. 令与边$\overline{p_ip_j}$相邻接的两个三角形分别为 $p_ip_jp_k$ 和 $p_ip_jp_l$
4. 在 $\mathcal{T}$ 中删除$\overline{p_ip_j}$，然后替之以$\overline{p_kp_l}$
5. **return** $\mathcal{T}$

这个算法迟早会终止，为什么呢？由〖观察结论 9.3〗可知，该算法中的每一轮迭代之后，$\mathcal{T}$的角度向量都会有所增长。对于任一集合P，可能的三角剖分只有有限个，因此这样一个单调变化的过程必然会终止。一旦算法终止，得到的结果必然是一个合法三角剖分。然而，尽管这个算法必然会终止，但是却需要经过很长的时间，因此价值不大。不过，我们还是暂且给出这个算法，因为后面将需要用到一个类似的函数。现在，让我们来看完全不同的（或者更准确地说，表面上看来完全不同的）另一个问题。

9.2 Delaunay 三角剖分

设P为由平面上n个点——有时也称之为基点（site）——组成的一个集合。你应该还记得第 7 章所讨论的Voronoi图。P的Voronoi图是平面的一个子区域划分，其中包含n个子区域，分别对应于P中的各个基点——任一基点$p \in P$所对应的子区域，由平面上以p为最近基点的所有点组成。

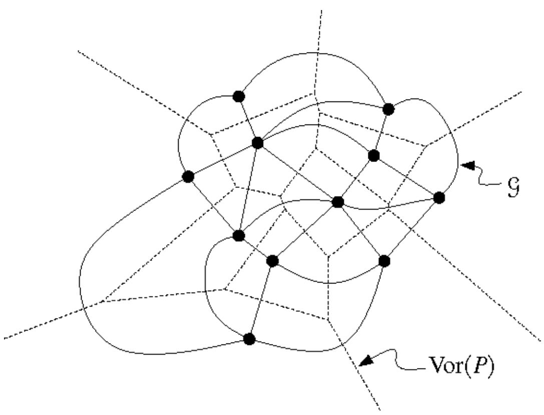


图9-10 Vor(P)的对偶图

P的Voronoi图，记作Vor(P)。与基点p相对应的子区域，称为p的Voronoi单元，记作$\mathcal{V}$(p)。本节将

要研究Voronoi图的对偶图。这个图记作$\mathcal{G}$。其中，对应于每一个Voronoi单元（或者，也可以说是对应于每一个基点），各有一个节点；若两个单元之间公用一条边，则在这两个单元各自对应的节点之间，联接一条弧（arc）。请注意，这就意味着，对应于Vor(P)中的每一条边，$\mathcal{G}$中都有一条弧与之对应。正如你可以从 图 9-10 中所看到的，在$\mathcal{G}$的所有有界面与Vor(P)中的所有顶点之间，存在一个一一对应关系。

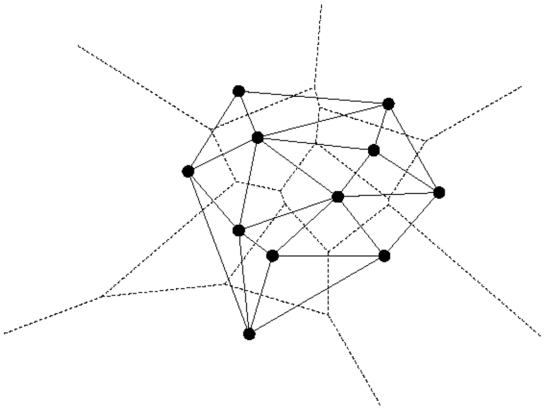


图9-11 Delaunay图$\mathcal{DG}$(P)

考察$\mathcal{G}$的如下直线嵌入（straight-line embedding）：其中，与Voronoi单元$\mathcal{V}$(p)对应的节点，用点p来实现；而联接于$\mathcal{V}$(p)和$\mathcal{V}$(q)之间的弧，则用线段$\overline{pq}$来实现（如 图 9-11 所示）。这一直线嵌入，称作P的Delaunay图（Delaunay graph），记作$\mathcal{DG}$(P)。（虽然这个名字听起来法国味道十足，但是Delaunay图却与“法国的画家”毫不相干。实际上，它是由数学家Boris Nikolaevich Delone而得名的。这位数学家的名字原文写出来大致为“Дедоне”，翻译成英文，本来应该是“Delone”。然而，在他的那个年代，法语和德语才是通用的科技语言，而他的这项成果则是用法语发表的，因此时至今日，人们更加熟悉的反倒是他用法语翻译的这个名字。）点集的Delaunay图，具有若干令人惊奇的特性。首先，它总是一个平面图——亦即，该直线嵌入中的任何两条边都不会相互跨越[①]。

〖定理 9.5〗

任何平面点集的 Delaunay 图都是一个平面图。

〖证明〗

为证明这一结论，需要利用Voronoi图的一个特性，该特性已经在 〖定理 7.4〗(ii)中被指出来了。为完整起见，此处将借助Delaunay图的概念，把这一特性复述一遍：

如 图 9-12 所示，$\overline{p_ip_j}$是Delaunay图$\mathcal{DG}$(P)中的一条边，当且仅当存在这样一个闭

① 即不相交于线段的内部。——译者

圆盘（closed disc）C_{ij}：它的边界经过p_i和p_j，而且其中不包含来自P的任何其它基点[①]。（当然，这样一个圆盘的中心必然落在$\mathcal{V}(p_i)$与$\mathcal{V}(p_j)$之间的公共边上。）

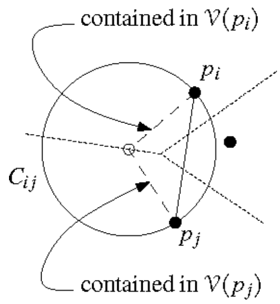


图9-12 每对基点及其共同边界上任一点所确定的圆，内部必然是空的

由顶点 p_i、p_j 以及 C_{ij} 的中心确定的那个三角形，记作 t_{ij}。可以看出，t_{ij} 的联接于 p_i 和圆心 C_{ij} 之间那条边，必然完全落在 $\mathcal{V}(p_i)$内；p_j 也有类似的性质。现在，任取 $\mathcal{DG}$(P)中的另一条边$\overline{p_k p_l}$，参照 C_{ij} 和 t_{ij} 的定义方法，也可以定义出圆盘 C_{kl} 以及三角形 t_{kl}。

假若本定理的断言不成立，即$\overline{p_i p_j}$与$\overline{p_k p_l}$相交。既然 p_k 和 p_l 必然都落在圆盘 C_{ij} 之外，故必然也落在三角形 t_{ij} 之外。这就意味着，在围成 t_{ij} 的三条边中，与 C_{ij} 的圆心相关联的那两条边必有其一与$\overline{p_k p_l}$相交。同理，在围成 t_{kl} 的三条边中，与 C_{kl} 的圆心相关联的那两条边也必有其一与$\overline{p_i p_j}$相交。于是，若考虑在 t_{ij} 的边界上、与 C_{ij} 的圆心相关联的那两条边，以及在 t_{kl} 的边界上、与 C_{kl} 的圆心相关联的那两条边，则前两条边中的某一条必然与后两条边中的某一条相交。然而，这种情况是不可能的——因为这两条边必然分别完全落在两个不同的 Voronoi 单元之内。 ❑

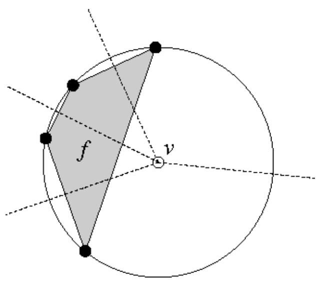


图9-13 Delaunay图中同一张面的各段边界，分别对应于Voronoi图中与同一个Voronoi顶点相关联的各条Voronoi边

P的Delaunay图，是Voronoi图的对偶图的一个直线嵌入。正如我们在此前已经注意到的，Vor(P)中的每个顶点，都分别对应于Delaunay图中的某张面。而在Delaunay图中围成每张面的各边，分别对

[①] 需要强调的是，由于这个圆盘是闭的，这句话意味着其它的基点既不能落在圆盘的内部（interior），也不能落在其边界（即对应的圆周）上。——译者

应于Voronoi图中与某个Voronoi顶点相关联的各条Voronoi边（如 图 9-13 所示）。具体而言，在Vor(P)中，若基点$p_1, p_2, p_3, \ldots, p_k$各自对应的（共k个）Voronoi单元有一个共同的顶点v，则在$\mathcal{DG}$(P)中，与v相对应的面f的各个顶点，必然就是$p_1, p_2, p_3, \ldots, p_k$。由〖定理 7.4〗(i)可知，在这种情况下，点$p_1, p_2, p_3, \ldots, p_k$必然散落在某个以v为中心的圆周上——因此可以看出，f不仅是一个k-多边形，而且它必然还是凸的。

如果 P 中各点是随机分布的，任何四点恰好共圆的可能性就会很小。任何集合，只要其中没有任何四点共圆，（在本章中）我们就称它是处于一般性位置的（in general position）。若 P 的确处于一般性位置，则在其对应的 Voronoi 图中，每个顶点的度数必然都是 3。于是，$\mathcal{DG}$(P)中的每一张有界面都必然是三角形。我们之所以经常将 $\mathcal{DG}$(P)称作“Delaunay 三角剖分”（Delaunay triangulation），原因正在于此。然而，在此我们还是应该更为谨慎一些，姑且将 $\mathcal{DG}$(P)称作 P 的“Delaunay 图”（Delaunay graph）。至于 Delaunay 三角剖分，我们对其有另一番定义：以 Delaunay 图为基础，通过引入联边而得到的一个三角剖分。既然 $\mathcal{DG}$(P)中的每张面都是一个凸集，（通过上述方法，）这样一个三角剖分就可以很容易地得到。需要注意的是，P 的 Delaunay 三角剖分是唯一确定的，当且仅当 $\mathcal{DG}$(P)本身已经是一个三角剖分了——也就是说，P 是处于“一般性位置”的。

至此，已经可以借助Delaunay图的概念，对关于Voronoi图的〖定理 7.4〗重新表述如下：

〖定理 9.6〗

设 P 为任一平面点集。则在 P 的 Delaunay 图中：

(i) 三个点 $p_i, p_j, p_k \in P$ 同为某张面的顶点，当且仅当 p_i, p_j, p_k 外接圆的内部不含 P 中的任何点；

(ii) 两个点 $p_i, p_j \in P$ 同时与某条边相关联，当且仅当存在一个闭圆盘 C，除了 p_i 和 p_j 落在其边界上之外，该圆盘不包含 P 中其它的任何点。

根据〖定理 9.6〗，可以立即得出Delaunay三角剖分的如下特性：

〖定理 9.7〗

设 P 为平面上的任一点集，而 $\mathcal{T}$ 为 P 的任一三角剖分。则 $\mathcal{T}$ 是 P 的 Delaunay 三角剖分，当且仅当在 $\mathcal{T}$ 中每个三角形的外接圆的内部，都不包含 P 中的任何点。

此前已经解释过：就高度差值之类的应用而言，为了得到“好的”三角剖分，也就是要使其对应的角度向量尽可能地大。接下来，我们就将对 Delaunay 三角剖分的角度向量作一考察。为此需要做些迂回——首先来考察所有的合法三角剖分。

〖定理 9.8〗

设 P 为平面上的任一点集。则$\mathcal{T}$是 P 的一个合法三角剖分，当且仅当$\mathcal{T}$是 P 的 Delaunay 三角剖分。

〖证明〗

首先，根据定义可以立即证明：任何 Delaunay 三角剖分都必然是一个合法三角剖分。

接下来，将通过反证法证明：任何合法三角剖分也必然是一个 Delaunay 三角剖分。假设存在 P 的某个合法三角剖分 $\mathcal{T}$，它不是一个 Delaunay 三角剖分。

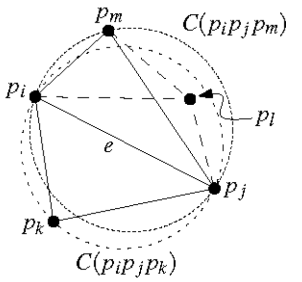


图9-14 非Delaunay三角剖分中，必然存在内部非空的圆$C(p_ip_jp_k)$

于是如 图 9-14 所示，根据 〖定理 9.6〗，必存在三角形$p_ip_jp_k$，在其外接圆$C(p_ip_jp_k)$的内部，包含有另一个点$p_l \in P$。三角形$p_ip_jp_k$的任一条边，都可以与点p_l共同确定一个三角形；其中，必有一个三角形与$p_ip_jp_k$不相交。不妨令之为$p_ip_jp_l$——也就是说，该三角形与三角形$p_ip_jp_k$的公共边为$e := \overline{p_ip_j}$。当然，这样的三角形$p_ip_jp_k$在$\mathcal{T}$中可能同时存在多个，如果是这样，就取其中使得角度$\measuredangle p_ip_lp_j$最大的那个。请注意：三角形$p_ip_jp_k$的三条边，将圆$C(p_ip_jp_k)$分割成四个部分：三角形$p_ip_jp_k$本身，以及另外三个拱形区域。现在，考察（在$\mathcal{T}$中）与$p_ip_jp_k$相邻于边e的那个三角形$p_ip_jp_m$。既然$\mathcal{T}$是合法的，则e必然是合法的。由 〖引理 9.4〗，p_m不可能落在$C(p_ip_jp_k)$的内部。因此，$C(p_ip_jp_k)$中以边e为底的那个拱形区域，必然被$p_ip_jp_m$的外接圆$C(p_ip_jp_m)$完全覆盖。于是，必然有$p_l \in C(p_ip_jp_m)$。此时，要么p_l与p_i分别处于直线$\overline{p_jp_m}$的两侧，要么p_l与p_j分别处于直线$\overline{p_ip_m}$的两侧。不失一般性地，假定为前一种情形。此时，根据Thales定理，必有$\measuredangle p_jp_lp_m > \measuredangle p_ip_lp_j$——这与$(p_ip_jp_k, p_l)$的角度最大性不合。 ❑

既然任一角度最优的三角剖分都必合法，故由〖定理 9.8〗可得出推论：P的任一角度最优的三角剖分，必是P的一个Delaunay三角剖分。当P处于一般性位置时，合法三角剖分是唯一存在的——它就是唯一的那个角度最优的三角剖分，即与Delaunay图完全吻合的那个唯一的Delaunay三角剖分。若P不是处于一般性位置，则在Delaunay图基础上的任何一个三角剖分，都是合法的。所有的这些Delaunay三角剖分，并不都是角度最优的。尽管如此，它们的角度向量也相差不大。而且，根据Thales

定理可以证明：对一组共圆的点，任何三角剖分中的最小角都是相等的。这就是说，最小角的大小与具体的三角剖分无关。因此，在同一Delaunay图的基础上继续进行三角剖分，无论得到哪个Delaunay三角剖分，其最小角总是相等的。这可以总结为如下定理：

〖定理 9.9〗

设 P 为任一平面点集。P 的任一角度最优的三角剖分，必是 P 的一个 Delaunay 三角剖分。此外，在 P 的所有三角剖分中，Delaunay 三角剖分使最小角达到最大。

9.3 构造 Delaunay 三角剖分

由以上分析可知，就我们的应用目标（比如根据一个采样点集 P 构造出一个多面式地形，以近似对应的地形）而言，P 的 Delaunay 三角剖分的确是一种适宜的三角剖分。其原因在于，Delaunay 三角剖分可使其中的最小角最大化。现在的问题是，如何才能构造出一个 Delaunay 三角剖分呢？

第 7 章介绍过构造点集P的Voronoi图的方法。一旦得到Vor(P)，就可以很容易得到Delaunay图 $\mathcal{DG}$(P)；接下来，只要对其中包含多于三个顶点的面做进一步三角剖分，即可最终得到一个Delaunay三角剖分。本节将另辟蹊径，采用随机增量式算法来直接计算Delaunay三角剖分。在第 4 章中解决线性规划问题时，以及在第 6 章中解决点定位问题时，这类算法都曾获得过成功。

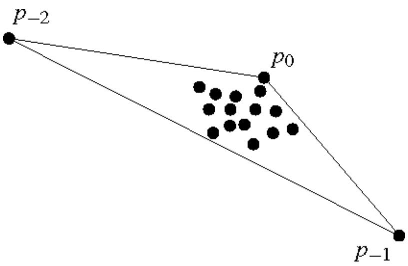


图9-15 足够大的包围三角形

第 6 章中，首先用一个足够大的矩形，将整个场景都包括进去。这样会有很多便利之处，比如，可以避免无界梯形（trapezoid）之类的麻烦。按照同样的思路，此处也首先用一个足够大的三角形将整个点集P包围起来（如 图 9-15 所示）。为此，需要引入两个辅助点p_{-1}和p_{-2}，它们与P中的最高点联合构成的三角形，将包含所有的点。也就是说，我们计算的是P∪{p_{-1}, p_{-2}}的（而不是P的）Delaunay三角剖分。这个三角剖分一旦构造出来，只需将p_{-1}、p_{-2} 以及与它们关联的各边删去，即可得到P的Delaunay三角剖分。为此，我们所选择的p_{-1}和p_{-2}必须相距足够远，才不致于对P的Delaunay三角剖分中的任何三角形有所影响。尤其必须保证的一点是，它们不能落在P中任何三点的外接圆内。具体的实现细节，将在稍后介绍；现在，还是先来看看算法本身。

这是一个随机增量式算法。我们按随机次序逐一引入各点，整个过程中，都要维护并更新一个与当前点集对应的Delaunay三角剖分。考虑引入点p_r时的情况。首先，要在当前的三角剖分中，确定p_r落在哪个三角形内（具体方法稍后将介绍）。然后，将p_r与该三角形的三个顶点分别联接起来，生成三条边。倘若p_r碰巧落在三角剖分的某条边e上，就需要找到与e关联的那两个三角形，然后将p_r与对顶的那两个顶点分别联接起来，生成两条边。这两种情况的处理方法如 图 9-16 所示。

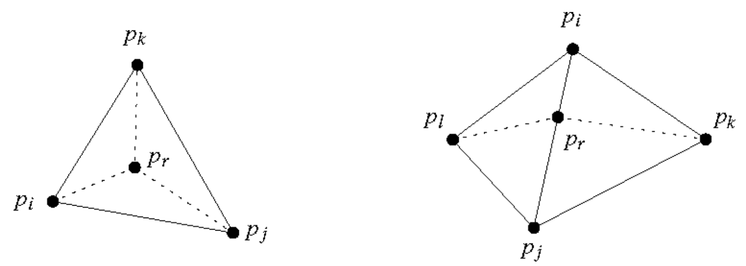


图9-16 引入点p_r时可能的两种情况：p_r落在某个三角形内部（左），p_r恰好落在某条边上（右）

如此又得到了一个三角剖分，但它不见得是一个 Delaunay 三角剖分——因为，在引入点 p_r之后，原来的某些边可能不再合法。为消除这些不合法性，需要针对每一条可能的非法边，调用一次子函数 LEGALIZEEDGE。这个子函数通过边翻转操作，将所有的非法边转换为合法边。以下首先对主算法做一准确的描述，然后再讨论这种转换的具体细节。为便于分析，不妨假定集合 P 由 n+1 个点组成。

算法 DELAUNAYTRIANGULATION(P)

输入： 由平面上 n+1 个点组成的一个集合 P

输出： P 的一个 Delaunay 三角剖分

1. 令 p_0 为 P 中依字典序最高的点，亦即，y-坐标最大的多个点中最靠右的那个
2. 在 $\mathbb{R}^2$ 中选取相距足够远的点 p_{-1} 和 p_{-2}，将 P 完全包含于三角形 $p_0p_{-1}p_{-2}$ 之中
3. 将 $\mathcal{T}$ 初始化为单独的一个三角形 $p_0p_{-1}p_{-2}$
4. 随机地选取 P\{p_0}中各点的一个次序：$p_1, \ldots, p_n$
5. **for** r ← 1 **to** n
6. 　　**do**(* 将 p_r 插入到 $\mathcal{T}$ 中 *)
7. 　　　　找到 p_r 所在的三角形 $p_ip_jp_k \in \mathcal{T}$
8. 　　　　**if** (p_r 落在三角形 $p_ip_jp_k$ 的内部)
9. 　　　　　　**then** 分别将 p_r 与三角形 $p_ip_jp_k$ 的三个顶点联接起来
　　　　　　　　(* 生成三条边，从而将三角形 $p_ip_jp_k$ 一分为三 *)
10. 　　　　　　　　LEGALIZEEDGE(p_r, $\overline{p_ip_j}$, $\mathcal{T}$)
11. 　　　　　　　　LEGALIZEEDGE(p_r, $\overline{p_jp_k}$, $\mathcal{T}$)

```
12.             LEGALIZEEDGE(p_r, p_k p_i, T)
13.         else (* p_r 正好落在三角形 p_i p_j p_k 的某一条边（不妨设为 p_i p_j）上)
14.             将 p_r 分别与 p_k 以及与 p_i p_j 关联的另一三角形的第三个顶点 p_l 联接起来
                (* 从而将与 p_i p_j 相关联的那两个三角形划分成四个三角形 *)
15.             LEGALIZEEDGE(p_r, p_i p_l, T)
16.             LEGALIZEEDGE(p_r, p_l p_j, T)
17.             LEGALIZEEDGE(p_r, p_j p_k, T)
18.             LEGALIZEEDGE(p_r, p_k p_i, T)
19.     将点 p_-1、p_-2 以及与之关联的所有边从 T 中剔除掉
20.     return (T)
```

接下来将要讨论的问题是：在第 8 行（或者第 13 行）得到了一个三角剖分之后，应该如何将其转换为一个Delaunay三角剖分。由〖定理 9.8〗可知，一个三角剖分是Delaunay三角剖分，当且仅当其中的所有边都是合法的。按照算法LEGALTRIANGULATION的原则，不断地对非法边实施翻转操作，直到重新回到一个合法三角剖分。只有一个问题尚待解决：在点p_r插入之后，哪些边有可能会变得非法？如 图 9-17 所示，我们注意到，原来的任何一条合法边$\overline{p_ip_j}$，只有在与其相关联的（最多两个）三角形之一发生变化时，才有可能会成为一条非法边。因此，我们只需检查新生成的那些三角形（的各边）。这项工作是由子程序LEGALIZEEDGE来完成的，它会对有关的各边进行检查，若有必要，则进行边翻转。每翻转一条边之后，可能又会进而使得其它的某些边变得非法。对于所有可能的新非法边，LEGALIZEEDGE都要递归地调用自己，逐一进行核查。

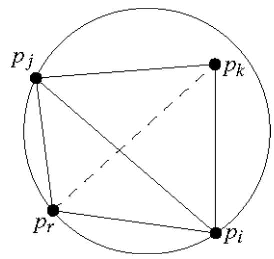


图9-17 只有在与之关联的三角形发生变化时，原先的合法边才可能转为非法边

```
算法 LEGALIZEEDGE(p_r, p_i p_j, T)
```

```
1.  (* pr 为插入的点，pipj 为 T 中可能需要翻转的一条边 *)
2.  if (pipj 是非法的)
3.      then 令 pipjpk 为沿着边 pipj 与 prpipj 相邻的三角形
4.          将原来的边 pipj 替换成边 prpk      (* 翻转 pipj *)
5.          LEGALIZEEDGE(pr, pipk, T)
6.          LEGALIZEEDGE(pr, pjpk, T)
```

其中，第 2 行要检查某条边的合法性。通常，利用〖引理 9.4〗的结论即可完成这一测试。然而在这里，由于特殊点p_{-1}和p_{-2}的存在，情况要略微复杂一些。这一问题将在稍后讨论；我们首先来证明该算法的正确性。

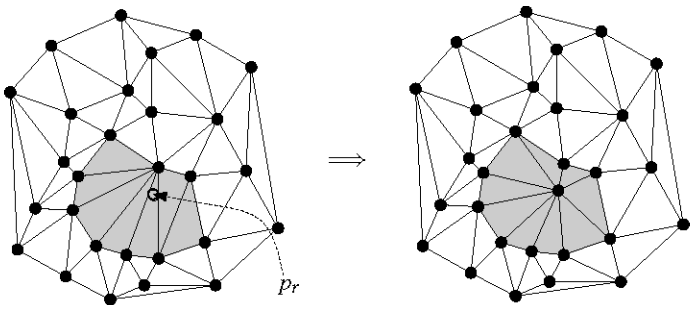


图9-18 新生出的每一条边，都必然与p_r相关联

为了保证该算法的正确性，需要证明：LEGALIZEEDGE子程序的所有调用都返回之后，将不会再有任何的非法边。从LEGALIZEEDGE的代码可以清楚地看出，由于p_r的插入而新生出来的每一条边，都必然与p_r相关联。这一点也可以从 图 9-18 中看出：在原有的某些三角形被销毁之后，新生出来的三角形用灰色表示。至关重要的一个观察结论（将在后面得到证明）是：所有新的边都是合法的——因此，并不需要对它们进行检查。此前我们已经注意到了：任何一条边若（从合法）变成非法，则与之相关联的（至多两个）三角形中必有其一发生了变化。综合这两个观察结论可知：所有可能变为非法的边，都必然会接受该算法的检查。也就是说，该算法是正确的。需要指出的是：与算法LEGALTRIANGULATION一样地，该算法也不致于陷入无限的死循环——这是因为，每经过一次翻转，三角剖分的角度向量总是会单调地增长。

〖引理 9.10〗

无论是由算法 DELAUNAYTRIANGULATION 生成的边，还是在插入点 p_r 的过程中生成的边，都是$\Omega \cup \{p_1, \ldots, p_r\}$的 Delaunay 三角剖分中的边。

〖证明〗

首先，考察将三角形 $p_ip_jp_k$（可能还有三角形 $p_ip_jp_l$）分割开来的边$\overline{p_rp_i}$、$\overline{p_rp_j}$、$\overline{p_rp_k}$（也可能还有$\overline{p_rp_l}$）。既然在插入点 p_r 之前，$p_ip_jp_k$ 曾经是 Delaunay 三角剖分中的一个三角形，故对于任何的 $t < r$，点 p_t 都不可能落在 $p_ip_jp_k$ 的外接圆 C 内。我们可以将 C 收缩为另一个圆 C'，这个圆包含于 C 之中，而且穿过 p_i 和 p_r。既然 $C' \subset C$，则 C'（的内部）必然是空的。这就说明，在插入点 p_r 之后，$\overline{p_rp_i}$必然是 Delaunay 图中的一条边。同样的道理，$\overline{p_rp_j}$和$\overline{p_rp_k}$也具有这一性质。（如果还有$\overline{p_rp_l}$，亦是如此。）

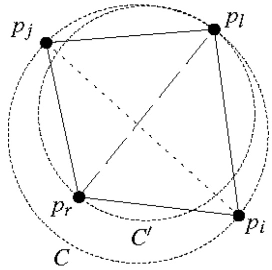


图9-19 经翻转操作后，边$\overline{p_ip_j}$被替换为边$\overline{p_rp_l}$

接下来，考察被LEGALIZEEDGE翻转的任一条边。如 图 9-19 所示，每经过一次这样的翻转操作，都会将某个三角形$p_ip_jp_k$的边$\overline{p_ip_j}$替换为与p_r相关联的另一条边$\overline{p_rp_l}$。在插入点p_r之前，$p_ip_jp_k$是一个Delaunay三角形，而且，p_r落在该三角形的外接圆之内（否则$\overline{p_ip_j}$就不会是非法的）。因此，我们总是能够将该外接圆收缩为另一个圆，使得除了p_r和p_l落在其边界上之外，这个圆的内部是空的。这样，在此次插入之后，$\overline{p_rp_l}$必然是Delaunay图中一条边。 ❑

以上证明了算法的正确性。下面，还需要就以下两个重要步骤的实现方法做一详细描述：在算法 DELAUNAYTRIANGULATION 的第 7 行，如何才能找到 p_r 所处的那个三角形？在子函数 LEGALIZEEDGE 第 2 行的测试中，如何才能妥善地处理点 p_{-1} 和 p_{-2}？首先回答前一问题。

为了确定p_r落在哪个三角形之中，可以模仿第 6 章曾经使用过的一种方法：在构造Delaunay三

角剖分的过程中，我们同时也构造一个点定位结构$\mathcal{D}$——它是一幅有向无环图（directed acyclic graph）。$\mathcal{D}$中的各匹叶子，分别对应于当前三角剖分$\mathcal{T}$中的各个三角形，而且在这些叶子节点与对应的三角形之间，我们也的确会维护一些指针，使它们互相指向对方。$\mathcal{D}$中的每个内部节点，都对应于曾经在此前某个阶段的三角剖分中存在过的某个三角形，只不过到了现在，这个三角形已经被销毁了。这个点定位结构的构造方法如下。在第 3 行，我们将$\mathcal{D}$初始化为只含有一个叶子节点的一幅 DAG——这个节点对应于三角形$p_0p_{-1}p_{-2}$。

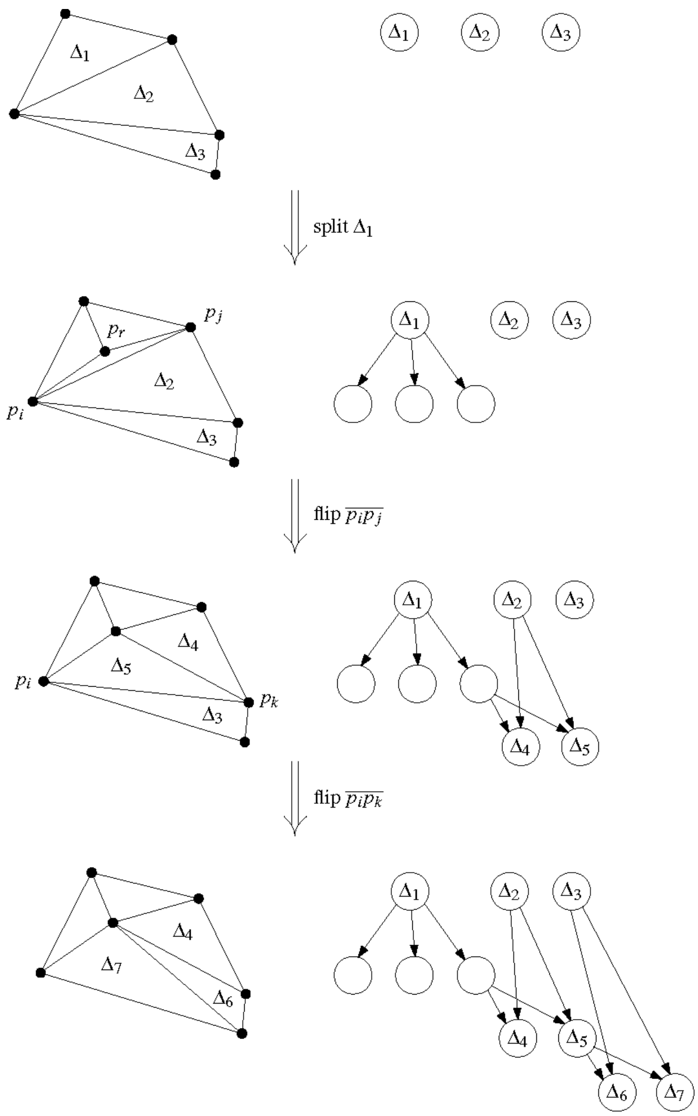


图9-20 将点p_r插入到三角形Δ_1中时，数据结构$\mathcal{D}$的相应变化（本图忽略了$\mathcal{D}$中没有发生变化的部分）

现在，假设在算法执行过程的某一步，我们将当前三角剖分中的某个三角形$p_ip_jp_k$进一步细分为三个（或者两个）三角形。与这个操作相对应地，结构$\mathcal{D}$中将会增加三匹（或者两匹）叶子；而原先对应于三角形$p_ip_jp_k$的那匹叶子，将成为一个内部节点；这个内部节点通过若干个指针，指向新近生成的这三匹（两匹）叶子。通过一次边翻转操作，可以将原来的两个三角形$p_kp_ip_j$和$p_ip_jp_l$替换为新的

三角形$p_kp_ip_l$和$p_kp_lp_j$，此时的处理方法也与之类似——对应于这两个新的三角形，分别生成一匹叶子，并且要通过指针，从原来与$p_kp_ip_j$和$p_ip_jp_l$对应的节点分别指向新的这两匹叶子。图 9-20 中所显示的，就是在引入一个新的点之后，结构$\mathcal{D}$随之发生的变化。需要注意的是，在将原来的某匹叶子转换为一个内部节点时，需要从该节点发出的指针最多不会超过 3 个。

借助于结构 $\mathcal{D}$，在将下一点 p_r 引入到三角剖分中的时候，可以按照如下方法确定其位置。从 $\mathcal{D}$ 的根节点（即对应于三角形 $p_0p_{-1}p_{-2}$ 的那个节点）开始。只要依次检查这个根节点的三个孩子节点，就可以确定 p_r 落在其中的那个三角形之中；然后，我们就转到与这个三角形相对应的那个孩子节点。接下来，逐一检查这个节点的各个孩子节点，进而转到其中包含 p_r 的一个（子）三角形。如此进行下去，直到到达 $\mathcal{D}$ 的某匹叶子。这匹叶子所对应的，就是在当前三角剖分中包含 p_r 的那个三角形。因为从任何节点发出的指针不会超过 3 个，因此这一查找过程所需要的时间，将线性正比于查照路径的长度——或者换而言之，线性正比于结构 $\mathcal{D}$ 中包含 p_r 的三角形总数。

现在，只剩下最后一个技术细节尚未交待——如何选取合适的 p_{-1} 和 p_{-2}？又如何实现对一条边合法性的测试？一方面，既然不希望由于 p_{-1} 和 p_{-2} 的存在而对 P 的 Delaunay 三角剖分造成任何影响，它们就必须相距足够远；而另一方面，我们也不希望为此使用巨大的坐标值。因此在这里，只是符号式地（symbolically）看待这些点——无需给它们赋予实际的坐标，而修改点定位与鉴别合法边的测试算法，使得其效果等同于这些点相距足够远。

以下，对于任意一对点 $p := (x_p, y_p)$和 $q := (x_q, y_q)$，若 $y_p>y_q$，或者 $y_p = y_q$ 且 $x_q > x_p$，则称“p 高于 q”。依此定义，即可在 P 中各点之间确定一个字典序。

分别取位于整个点集 P 之下、之上的一对水平线 l_{-1} 和 l_{-2}。假想地在 l_{-1} 上取 p_{-1}，沿 l_{-1} 向右移动 p_{-1}，直到它不再落在 P 中任何三个点的外接圆内，并使 P 中各点相对于 p_{-1} 的极角次序，与它们的字典序完全一致。然后，假想地在 l_{-2} 上取 p_{-2}， 沿 l_{-1} 向左移动 p_{-1}，直到它不再落在 $P\cup\{p_{-1}\}$中任何三个点的外接圆内，并且使得 $P\cup\{p_{-1}\}$中各点相对于 p_{-2} 的极角次序，与它们的字典序完全一致。

$P\cup\{p_{-1}, p_{-2}\}$的 Delaunay 三角剖分由四部分组成：P 的 Delaunay 三角剖分，p_{-1} 与 P 的右侧凸包上各点之间的联边，p_{-2} 与 P 的左侧凸包上各点之间的联边，以及 p_{-1} 与 P_{-2} 之间的联边$\overline{p_{-1}p_{-2}}$。P 中的最高点 p_0 和最低点，与 p_{-1} 和 p_{-2} 之间都有联边。

在点定位阶段，需要判断点 p_j 相对于有向直线$\overrightarrow{p_ip_k}$的位置。得益于以上 p_{-1} 和 p_{-2} 的选取方法，以下条件都是等价的：

- p_j 位于有向直线$\overrightarrow{p_ip_{-1}}$的左侧；
- p_j 位于有向直线$\overrightarrow{p_{-2}p_i}$的左侧；

- 按字典序，p_j 大于 p_i。

在判断某条边的合法性时，又该如何处理 p_{-1} 和 p_{-2} 呢？设待测试的边为 $\overrightarrow{p_ip_j}$，设与该边关联的两个三角形（如果都存在的话）的第三个顶点分别为 p_k 和 p_l。

- $\overrightarrow{p_ip_j}$ 为三角形 $p_0p_{-1}p_{-2}$ 的一条边。这类边必定合法。
- 下标 i、j 和 k 均非负。这是最常见的情况，参与测试的这些点都不是当做符号来处理的。因此，$\overrightarrow{p_ip_j}$ 非法，当且仅当 p_l 落在 p_i、p_j 和 p_k 的外接圆内。
- 所有其余的情况。这些情况下，$\overrightarrow{p_ip_j}$ 合法当且仅当 p_l 落 $\min(k, l) < \min(i, j)$。

最后一种情况还需进一步说明。其中，$\overrightarrow{p_ip_j}$ 即 $\overrightarrow{p_{-1}p_{-2}}$ 的情况，属于第一类情况，故而可以假定下标 i 和 j 中至多一个为负。另一方面，鉴于 p_k 和 p_l 之一必定就是刚刚插入的点 p_r，下标 k 和 l 中也至多有其一为负。

若四个下标中中有一个为负，则对应的点必落在另三个点外接圆之外，此时我们的方法正确。

否则，$\min(i, j)$ 和 $\min(k, l)$ 都是负的。再考虑到 p_{-2} 必落在 $P\cup\{p_{-1}\}$ 中任意三点的外接圆之外，说明我们的方法（在这一情况下）也是正确的。

9.4 分析

首先，我们要对该算法执行时数据结构的演变过程做一考察。这一过程中数据结构的总体变化量，也就是算法生成和销毁的三角形总数。在开始分析之前，需要引入两个记号：$P_r := \{p_1, \ldots, p_r\}$，以及 $\mathcal{DG}_r := \mathcal{DG}(\Omega \cup P_r)$。

〖引理 9.11〗

由算法 DELAUNAYTRIANGULATION 生成的三角形，总数目的期望值不超过 $9n + 1$。

〖证明〗

一开始，我们要以Ω中的点为顶点，生成单独的一个三角形。在算法的第 r 轮迭代中，我们要插入 p_r。为此，首先要对一个或者两个三角形进行细分，得到三个或者四个新的三角形。

无论如何，经过这一细分后生成的新边的数目总是确定的——具体讲，也就是 $\overline{p_rp_i}$、$\overline{p_rp_j}$、$\overline{p_rp_k}$ 和 $\overline{p_ip_j}$（或者是 $\overline{p_rp_l}$）。此外，在 LEGALIZEEDGE 过程中我们每翻转一条边，也会生成两个新的三角形。而且同样地，经过每次翻转操作，还会在 $\mathcal{DG}_r$ 中生成一条与 p_r 相关联的新边。总而言之：

在插入点 p_r 之后，若 $\mathcal{DG}_r$ 中与 p_r 关联的边数为 k，则我们所生成的三角形的数目不会超过 2(k-3) + 3 = 2k – 3。其实，k 也就是 p_r 在 $\mathcal{DG}_r$ 中的度数；记作 $\deg(p_r, \mathcal{DG}_r)$。

那么，就集合P可能的所有排列而言，p_r的度数的期望值等于多少呢？为了界定这一数值，我们需要再次使用曾经在第 4 章和第 6 章使用过的方法——后向分析。也就是说，在这一时刻，我们首先要固定集合P_r。我们需要界定的，是P_r中的一个随机成员——点p_r——的期望度数。根据〖定理 7.3〗，Delaunay图 $\mathcal{DG}_r$中至多含有 3(r+3) - 6 条边。将三角形$p_{-1}p_{-2}p_{-3}$的三条边从其中排除掉，P_r中各顶点度数的总和要小于 2 × [3(r+3) – 9] = 6r。这就是说，P_r中任一随机点的期望度数为 6。将上述分析结论综合起来，我们就可以这样来界定在第r步中生成三角形的数目：

$$
\begin{aligned}
& E[\text{第 r 步生成的三角形数目}] \\
& \leq E[2 \cdot \deg(p_r, \mathcal{DG}_r) - 3] \\
& = 2 \times E[\deg(p_r, \mathcal{DG}_r)] - 3 \\
& \leq 2 \times 6 - 3 \\
& = 9
\end{aligned}
$$

生成的三角形总数，等于最开始时生成的那个三角形 $p_{-1}p_{-2}p_{-3}$，以及随后每一步插入过程中所生成三角形的数目之和。故由期望的线性律可知：其期望值不超过 1 + 9n。 ❑

这样就得出了如下的最后结论：

〖定理 9.12〗

任意给定由平面上 n 个点组成的一个集合 P，我们都可以使用 $O(n)$的期望空间，在 $O(n\log n)$的期望时间内构造出 P 的 Delaunay 三角剖分。

〖证明〗

算法的正确性，在上面的讨论中已经得证。至于其空间复杂度，我们可以注意到：只有查找结构$\mathcal{D}$，才有可能占用超过线性规模的存储空间。然而，$\mathcal{D}$中的每一个节点，都对应于在该算法过程中生成的某一三角形。根据〖引理 9.11〗，其期望值为$O(n)$。

为了界定该算法的期望运行时间，我们暂且不计（第 6 行）点定位查询所消耗的时间。这样，该算法的运行时间，就正比于其生成的三角形总数。再一次可由〖引理 9.11〗得出结论：若不考虑用于点定位查询的时间，该算法的期望运行时间为$O(n)$。

现在，回过头来估计点定位查询所消耗的时间。每次在当前的三角剖分中确定点 p_r 的位置，所需要的时间都将正比于我们需要在 $\mathcal{D}$ 中访问的节点总数。而任何一个被访问到的节点，都对应于算法在此前所生成的、包含点 p_r 的某一三角形。在这些三角形中，除了最后（对应于一匹叶子的）那个之外，都满足下面三条性质：①它诞生于算法此前的某个阶段；②但后来又（通

过细分或翻转操作）被销毁了；③当然，它还必须包含点 p_r。因此，若对当前三角剖分中的三角形分别统计，则对点 p_r 进行定位所需的时间可以划分为两部分：前一部分为常数 $O(1)$，后一部分线性正比于满足上述三条性质的三角形总数。

三角剖分中的任何一个三角形 $p_ip_jp_k$ 若被销毁了，其原因不外乎以下两种之一：

- 插入点 p_l，该点位于三角形 $p_ip_jp_k$ 的内部（或者落在其边界上）；
- 通过边翻转操作，将三角形 $p_ip_jp_k$ 以及与之相邻的另一个三角形 $p_ip_jp_l$，替换为新的一对三角形 $p_kp_ip_l$ 和 $p_kp_jp_l$。

若是前一原因，则在插入点 p_l 之前，$p_ip_jp_k$ 原本就是一个 Delaunay 三角形。若是后一种情况，则要么 $p_ip_jp_k$ 是原来的一个 Delaunay 三角形，而且插入的点是 p_l，要么 $p_ip_jp_l$ 是原来的一个 Delaunay 三角形，而且插入的点是 p_k。如果 $p_ip_jp_l$ 是原来的一个 Delaunay 三角形，那么既然后来需要对边 $\overline{p_ip_j}$ 实施翻转操作，就必然意味着点 p_k 和 p_r 都落在 $p_ip_jp_l$ 的外接圆内。

无论是哪一种情况，只要三角形 $p_ip_jp_k$ 的确被访问到了，我们就总是可以将导致这次访问的原因，归于另一个 Delaunay 三角形Δ，而且Δ与 $p_ip_jp_k$ 同时被销毁掉了——也就是说，点 p_r 落在Δ的外接圆内。任意给定一个三角形Δ，由 P 中所有落在Δ外接圆之内的点构成的集合，记作 K(Δ)。根据前面的分析，在确定点 p_r 位置的过程中需要访问到某个三角形的原因，可以归于满足 $p_r \in K(\Delta)$ 的某个三角形Δ。不难看出，对于 K(Δ)中的任何一点，三角形Δ最多只能起到一次这种作用。因此，所有点定位查询所消耗的时间总量为：

$$O(n + \sum_{\Delta}\text{card}(K(\Delta))) \quad (9.1)$$

求和范围覆盖算法所生成的所有三角形。稍后将证明：该和式的期望值为 $O(n\log n)$。 ❑

接下来需要解决的问题，就是界定出各个集合K(Δ)的大小。若Δ是$\Omega \cup P_r$的Delaunay三角剖分中的一个三角形，card(K(Δ))的期望值应该等于多少呢？我们知道：在$r = 1$ 时，其期望值大致为n；而当$r = n$时，则应该是零。然而，介于这两个极端之间，情况又会如何呢？随机化（randomization）具有这样一点好处：通常它总是会在这两个极端之间进行“插值”。马上会闪入我们脑子里的一个直觉就是：既然P_r是一个随机样本，那么落在三角形$\Delta \in \mathcal{DG}_r$内部的点数就应该差不多是$O(n/r)$。这个直觉是正确的，然而千万不要掉以轻心——实际上，这一点并不见得对$\mathcal{DG}_r$中的所有三角形都成立。尽管如此，在计算表达式 (9.1)时，这一点看起来的确是成立的。

本节剩下的部分将就点集处于一般性位置的情况，对上述事实给出一个扼要的证明。对于一般化的情况，这一结论同样能够成立。不过，想要现在就给出证明，恐怕得费九牛二虎之力。因此，我们只好将这项工作推迟到下一节——在那里，我们将在更具一般性的条件下解决这一问题。

〖引理 9.13〗

对于任何处于一般性位置的点集 P，都必有

$$\sum_{\Delta} \text{card}(K(\Delta)) = O(n\log n)$$

其中的求和范围，覆盖由算法生成的所有 Delaunay 三角形Δ。

〖证明〗

既然 P 处于一般性位置，它的任一子集 P_r 也必然处于一般性位置。这就意味着：在插入点 p_r 之后所得到的三角剖分必然是唯一确定的，它就是 $\mathcal{DG}(\Omega\cup P_r)$。将 $\mathcal{DG}(\Omega\cup P_r)$中的三角形组成一个集合，记作 $\mathcal{T}_r$。根据这一定义，在第 r 轮迭代中生成的那些 Delaunay 三角形合起来，正好等于 $\mathcal{T}_r\backslash\mathcal{T}_{r-1}$。于是，就可以将我们准备界定的那个求和式重新表述为：

$$\sum_{r=1}^{n}\left(\sum_{\Delta\in\mathcal{T}_r\backslash\mathcal{T}_{r-1}} \text{card}(K(\Delta))\right)$$

对于每个点 q，我们都用 $k(P_r, q)$来表示满足 $q\in K(\Delta)$的三角形$\Delta\in\mathcal{T}_r$；此外，令 $k(P_r, q, p_r)$为既满足 $q \in K(\Delta)$，而且也与 p_r 相关联的那些三角形$\Delta \in \mathcal{T}_r$ 的数目。我们知道，在第 r 轮迭代中生成的任何一个 Delaunay 三角形，都必然与 p_r 相关联。因此就有：

$$\sum_{\Delta\in\mathcal{T}_r\backslash\mathcal{T}_{r-1}} \text{card}(K(\Delta)) = \sum_{q\in P\backslash P_r} k(P_r, q, p_r) \quad\text{...}\quad (9.2)$$

现在暂且将 P_r 固定。也就是说，我们将总是在“假定 P_r 等于某个固定集合 P_r^*”的前提下，相对于集合 P 的所有排列来考察其期望值。这样，$k(P_r, q, p_r)$的大小将取决于 p_r 的选取。一个三角形$\Delta\in\mathcal{T}_r$ 与一个随机点 $p\in P_r^*$相关联的概率，至多不过$\frac{3}{r}$，因此就得到：

$$E[k(P_r, q, p_r)] \le \frac{3\cdot k(P_r, q)}{r}$$

只要对所有的$q\in P\backslash P_r$进行求和，再应用式 (9.2)可得：

$$E\left[\sum_{\Delta\in\mathcal{T}_r\backslash\mathcal{T}_{r-1}} \text{card}(K(\Delta))\right] \le \frac{3}{r}\cdot \sum_{q\in P\backslash P_r} k(P_r, q) \quad\text{..}\quad (9.3)$$

$P\backslash P_r$ 中的任何一个点 q，都有均等的机会成为 p_{r+1}，因此有：

$$E[k(P_r, p_{r+1})] = \frac{1}{n-r}\cdot \sum_{q\in P\backslash P_r} k(P_r, q)$$

将这一等式代入不等式 (9.3)，就得到了：

$$E[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} card(K(\Delta))] \leq 3 \cdot (\frac{n-r}{r}) \cdot E[k(P_r, p_{r+1})]$$

那么，$k(P_r, p_{r+1})$又等于多少呢？它正是$\mathcal{T}_r$中满足$p_{r+1} \in K(\Delta)$的三角形Δ的数目。根据〖定理 9.6〗(i)的准则，这类三角形正是在插入点p_{r+1}之后将被销毁掉的那些三角形。因此，可进一步将上式写成：

$$E[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} card(K(\Delta))] \leq 3 \cdot (\frac{n-r}{r}) \cdot E[card(\mathcal{T}_r \setminus \mathcal{T}_{r+1})]$$

〖定理 9.1〗告诉我们，$\mathcal{T}_m$中所含三角形的总数正好是 2(m+3) - 2 - 3 = 2m + 1。故而，在插入点p_{r+1} 后，将被销毁掉的三角形的数目，正好要比新生成的三角形少两个。因此，上述求和式又可以进一步写成：

$$E[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} card(K(\Delta))] \leq 3 \cdot (\frac{n-r}{r}) \cdot E[card(\mathcal{T}_{r+1} \setminus \mathcal{T}_r) - 2]$$

需要指出的是，直到现在，我们一直都是将 P_r 看成一个固定的集合。这样，我们就可以针对满足 $P_r \subset P$ 的所有可能的 P_r，直接对这一不等式的两端分别进行平均；而且，若针对集合 P 的所有排列来计算期望值，这一不等式依然成立。

我们已经知道，在插入点 p_{r+1} 的过程中所生成三角形的数目，总是等于在 $\mathcal{T}_{r+1}$ 中与 p_{r+1} 相关联的边数；而且，这种边的期望数目等于 6。因此可以最终得到：

$$E[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} card(K(\Delta))] \leq 12 \cdot (\frac{n-r}{r})$$

最后，只需将这一不等式针对所有的 r 进行求和，本引理即可得证。 ❑

9.5 *随机算法框架

至此，读者应该已经在本书中见过了三个随机增量式算法：第一个用于解决第 4 章中的线性规划问题，另一个用于解决第 6 章中梯形图的计算问题，还有一个出现在本章，用来计算Delaunay三角剖分。(第 11 章还会介绍一个这样的算法。)这几个算法与出现在计算几何（computational geometry）文献中的其它此类算法一样，其工作原理如下。

假设待解决的问题是：根据一组输入的几何对象X，构造某种几何结构$\mathcal{T}$(X)（例如，给定平面上的一组点，构造与之对应的Delaunay三角剖分）。若利用随机增量式算法来解决这一问题，将按照随机的次序，逐个引入X中的各个对象；与此同时，还要对结构$\mathcal{T}$做动态的维护和更新。每插入一个

新的对象，算法都将首先确定，由于该对象的引入，结构$\mathcal{T}$中的哪些部分将会发生冲突，以至于需要进行调整——这一步称作“定位”（location）；然后，需要在这些位置对结构$\mathcal{T}$实施局部调整——这一步称作“更新”（update）。正是因为所有的随机增量式算法都是如此相似，所以对它们的分析方法也就大同小异。虽然针对不同的问题有不同的随机增量式算法，但即使你愿意不厌其烦地对每个算法都按部就班地进行一次分析，所得出的复杂度上界（upper bound）却都是一样。为了省去这些重复性工作，我们必须建立起一套公理框架（axiomatic framework），从各种随机增量式算法中提炼出其本质的共性。这一框架被称作构形空间（configuration space），利用这一框架，许多随机增量式算法的复杂度上界都可以直接得到。（不巧的是，“configuration space”一词已经在运动规划领域中被采用了，在那个领域里，该术语的含义与这里完全不同——具体请参见第 13 章。）本节将先对这一框架进行描述，然后再给出一个定理。对符合这一框架的任何随机增量式算法，我们都可以运用这一定理进行分析。例如，只要借助该定理，〖引理 9.13〗就可以立即得证；而且，这种方法无需假定P处于一般性位置。

所谓的一个构形空间，是一个四元组(X, Π, D, K)。其中，X 为问题的输入，由一组共有限个（几何）对象组成；我们用 n 来表示 X 的基数。集合 Π 中的每个元素，都称为一个构形（configuration）。最后，对 Π 中的每一个构形Δ，D 和 K 都为其指定了 X 的某个子集，分别记作 D(Δ)和 K(Δ)。我们说，集合 D(Δ)的各元素分别**定义**（define）了某个构形Δ，而集合 K(Δ)的各元素则与某个构形Δ发生**冲突**（conflict），或者说**毁灭**（kill）了构形Δ。集合 K(Δ)中所含元素的数目，称作构形Δ的**冲突规模**（conflict size）。(X, Π, D, K)必须满足下列条件：

- $d := \max\{\mathrm{card}(D(\Delta)) \mid \Delta \in \Pi\}$必须是一个常数。称作该构形空间的**最大度数**（maximum degree）。另外，由同一集合定义的构形，总数不得超过某个常数上界。
- 对于所有的$\Delta \in \Pi$，都有 $D(\Delta) \cap K(\Delta) = \varnothing$。

对于任一构形Δ，若 X 的某个子集 $S \subseteq X$ 将 D(Δ)包含于其中，却与 K(Δ)不相交，我们就说Δ是“在 S 上**活跃的**（active）”。对于 X 的任一子集 S，我们将在 S 上活跃的所有构形组成一个集合，记作 $\mathcal{T}$(S)。也就是说：

$$\mathcal{T}(S) := \{\Delta \in \Pi : D(\Delta) \subseteq S,\ \text{而且}\ K(\Delta) \cap S = \varnothing\}$$

这些活跃的构形，就构成了我们所希望构造的（几何）结构。更确切地说，我们的目标就是要计算出 $\mathcal{T}$(X)。在对这一抽象框架深入讨论之前，我们还是首先来回顾一下截至目前所遇到的各种几何结构，看看它们是否符合这一框架。

9.5.1 半平面求交

在这一问题中，输入集合X就是平面上的一组半平面。我们需要定义好Π、D以及K，使得$\mathcal{T}$(X)

恰好就是我们希望构造的东西——具体说，也就是X中所有半平面的公共交集（如 图 9-21 所示）。

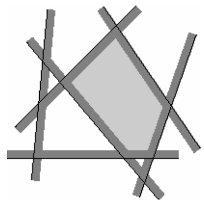


图9-21 一组半平面的公共交集

为此可以采用如下方法。首先，找出 X 中各半平面的边界线，这些直线之间的所有交点组成了构形集合 Π。对于其中的任一构形Δ ∈ Π，其**定义集**（defining set）D(Δ)由确定这一交点的那两条直线组成；而其**毁灭集**（killing set）K(Δ)，则由所有不含该交点的半平面组成。这样，无论是对于 X 的任一真子集 S ⊂ X，还是对于 S = X 本身，集合 T(S)都由 S 中各半平面的公共交集的所有顶点组成。

9.5.2 梯形图

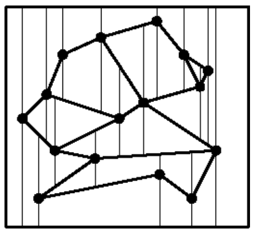


图9-22 梯形图

就这一问题而言，输入集X是平面上的一组线段。而集合Π中的每一个构形就是一个梯形，这种梯形必然出现在X的某个子集S ⊆ X所对应的梯形图中（如 图 9-22 所示）。对于任一构形Δ，其定义集 D(Δ)由确定Δ所必需的那些线段组成；而梯形Δ的毁灭集 K(Δ)，则由那些与Δ相交的线段组成。按照这样的定义，集合T(S)恰好就是由S所对应的梯形图中的所有梯形组成的。

9.5.3 Delaunay 三角剖分

如 图 9-23 所示，输入集X为平面上处于一般性位置的一组点。而集合Π中的每一个构形Δ都是一个三角形，它由X中（不共线的）某三个点确定。定义集 D(Δ)是一个点集，其中的元素也就是Δ的各个顶点；而毁灭集 K(Δ)则由落在Δ外接圆内的那些点组成。T(S)由若干三角形构成，根据〖定理 9.6〗，这些三角形恰好组成了S的那个唯一的Delaunay三角剖分。

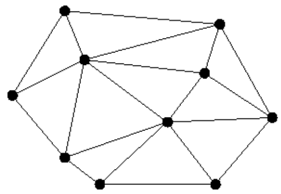


图9-23 Delaunay三角剖分

正如此前所交待过的，我们的目标是构造出结构 $\mathcal{T}(X)$。若采用随机增量式算法来实现这一构造过程，我们将首先计算出 X 中所有对象的一个随机排列 $x_1, \ldots, x_n$，然后再按照这一次序逐一引入这些对象；在此过程中，我们还要动态地维护 $\mathcal{T}(X_r)$，其中的 $X_r := \{x_1, \ldots, x_r\}$。之所以可以这样做，是因为构形空间具有这样一个基本的性质：只要在局部对某个构形Δ进行检查，就可以判断出Δ究竟是否属于 $\mathcal{T}(X_r)$——为此，只需找到Δ的定义集和毁灭集。具体来说，无论 X_r 中的各个对象是按照何种次序被引入的，最终得到的 $\mathcal{T}(X_r)$都是一样的。例如，某个三角形Δ属于 S 的 Delaunay 三角剖分，当且仅当Δ的顶点均来自于 S，而且 S 中的任何点都没有落在Δ的外接圆内。

在对随机增量式算法进行分析的时候，通常首先要做的，就是为该结构的期望变化量建立一个（上）界。这类例子很多，比如〖引理 9.11〗。下面这则定理的作用就是完成这项工作，不过，它是在抽象出来的构形空间框架中来完成这项工作。

〖定理 9.14〗

设(X, Π, D, K)为一个构形空间，取$\mathcal{T}$和 X_r 定义如上。于是，包含于 $\mathcal{T}(X_r) \setminus \mathcal{T}(X_{r-1})$中的构形的数目，期望值不会超过

$$\frac{d}{r} \cdot E[\mathrm{card}(\mathcal{T}(X_r))]$$

其中，d 为该构形空间的最大度数。

〖证明〗

在此前的几种情况中，都要对结构的变化量做出界定；这里亦是如此，我们也要采用后向分析的方法——也就是说，我们试图统计的不是在将 x_r 插入到 X_{r-1} 中之后而生出的构形的数量，而是在 X_r 中删除 x_r 之后随之消失的构形的数量。为此，不妨令 X_r 为 X 的某个固定子集 $X_r^* \subset X$，其基数等于 r。这样，我们的任务就是界定出，在将某个随机对象 x_r 从 X_r 中删除之后，$\mathcal{T}(X_r)$中随之消失的那些构形$\Delta \in \mathcal{T}(X_r)$的期望数目。按照$\mathcal{T}$的定义，这种Δ必然满足 $x_r \in D(\Delta)$。满足$\Delta \in \mathcal{T}(X_r)$和 $x \in D(\Delta)$的二元组(x, Δ)，至多只有 $d \cdot \mathrm{card}(\mathcal{T}(X_r))$个。由此可知：

$$\sum_{x \in X_r} \mathrm{card}(\{\Delta \in \mathcal{T}(X_r) \mid x \in D(\Delta)\}) \quad \leq \quad d \cdot \mathrm{card}(\mathcal{T}(X_r))$$

这样，在 X_r 中随机删除一个对象之后，随之消失的构形的期望数目，就应该等于 $\frac{d}{r} \cdot \text{card}(\mathcal{T}(X_r))$。为了得出这一结论，我们所取的 X_r 是 X 的一个基数为 r 的固定子集 $X_r^* \subset X$。为了进而得出一般性的上界，还需要考虑到所有可能的基数为 r 的子集，并对它们做平均——这样，就得到了 $\frac{d}{r} \cdot E[\text{card}(\mathcal{T}(X_r))]$。 ❑

在随机增量式算法的运行过程中，（几何）结构的变化总量会有多大？〖定理 9.14〗已经对此给出了一个一般性的上界。然而在另一方面，点定位的计算量又是多少呢？本章也就此给出了一个上界；而在其它的很多时候，都将需要得到一个与之形式相同结果。具体而言，我们的任务就是要得出下面这个和式的上界：

$$\sum_{\Delta} \text{card}(K(\Delta))$$

这里的求和范围，覆盖了由算法生成的所有构形Δ——任何一个这样的构形，都出现在（至少）某一个 $\mathcal{T}(X_r)$中。下面这则定理，给出了它的一个上界。

〖定理 9.15〗

设(X, Π, D, K)为一个构形空间，取 $\mathcal{T}$ 和 X_r 定义如上。于是，若考虑所有的 $\mathcal{T}(X_r)$（$1 \le r \le n$），并对至少出现在其中某一个 $\mathcal{T}(X_r)$之中的那些构形Δ进行求和 $\sum_{\Delta} \text{card}(K(\Delta))$，则该和式的期望值不会超过

$$\sum_{r=1}^{n} d^2 \cdot \left(\frac{n-r}{r}\right) \cdot \left(\frac{E[\text{card}(\mathcal{T}(X_r))]}{r}\right)$$

其中，d 为该构形空间最大度数。

〖证明〗

这里几乎可以照搬〖引理 9.13〗的证明过程。首先，将这一求和式整理为：

$$\sum_{r=1}^{n} \left(\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} \text{card}(K(\Delta)) \right)$$

然后，考察 $\mathcal{T}(X_r)$中满足 $y \in K(\Delta)$的构形Δ，将这类Δ的总数记作 $k(X_r, y)$；考察 $\mathcal{T}(X_r)$中同时满足 $y \in K(\Delta)$和 $x_r \in D(\Delta)$的构形Δ，将这类Δ的总数记作 $k(X_r, y, x_r)$。在插入 x_r 之后，随之生出的每一个新构形Δ都必然满足 $x_r \in D(\Delta)$。这就意味着：

$$\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} \mathrm{card}(K(\Delta)) = \sum_{y \in X \setminus X_r} k(X_r, y, x_r) \quad (9.4)$$

现在，将集合 X_r 固定住。这样，$k(X_r, y, x_r)$的期望值就取决于在 X_r 中对 x_r 的选取。对于 $\mathcal{T}(X_r)$ 中的任一构形Δ，$y \in \Delta$成立的概率不会超过$\frac{d}{r}$，因此就有：

$$E[k(X_r, y, x_r)] \le \frac{d \cdot k(X_r, y)}{r}$$

只要对所有的$y \in X \setminus X_r$，对该不等式求和，然后利用等式 (9.4)，就可以进一步得到：

$$E\Big[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} \mathrm{card}(K(\Delta))\Big] \le \frac{d}{r} \cdot \sum_{y \in X \setminus X_r} k(X_r, y) \quad (9.5)$$

另一方面，$X \setminus X_r$ 中的每一个 y 作为 x_{r+1} 出现的可能性都是相同的，于是就有：

$$E[k(X_r, x_{r+1})] = \frac{1}{n-r} \cdot \sum_{y \in X \setminus X_r} k(X_r, y)$$

将这个等式代入不等式 (9.5)，就得到了：

$$E\Big[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} \mathrm{card}(K(\Delta))\Big] \le d \cdot \left(\frac{n-r}{r}\right) \cdot E[k(X_r, x_{r+1})]$$

在插入 x_{r+1} 的下一次迭代中，$\mathcal{T}(X_r)$中的某些构形Δ将被销毁掉。现在可以看到，所谓的 $k(X_r, x_{r+1})$，正是这些将被销毁掉的构形的总数。这就是说，可以将上面最后那个不等式重新写成：

$$E\Big[\sum_{\Delta \in \mathcal{T}_r \setminus \mathcal{T}_{r-1}} \mathrm{card}(K(\Delta))\Big] \le d \cdot \left(\frac{n-r}{r}\right) \cdot E[\mathrm{card}(\mathcal{T}(X_r) \setminus \mathcal{T}(X_{r+1}))] \quad (9.6)$$

然而，与〖引理 9.13〗的证明稍有不同的是，这里不能直接通过在第r+1 轮迭代中生成的构形的数目，来界定在该轮迭代中被销毁掉的构形的数目——原因在于，在一个一般的构形空间中，这并不见得一定成立。因此，后面的证明将稍作变化。

首先请注意到，不等式 (9.6)的两边，都可以分别对所有可能的X_r进行平均；而且，在取遍X的所有排列之后，该不等式将依然成立。然后，再对所有r进行求和，并将其结果写成：

$$\sum_{r=1}^{n} d \cdot \left(\frac{n-r}{r}\right) \cdot \mathrm{card}(\mathcal{T}(X_r) \setminus \mathcal{T}(X_{r+1})) = \sum_{\Delta} d \cdot \left(\frac{n-[j(\Delta)-1]}{j(\Delta)-1}\right) \quad (9.7)$$

其中，右侧的求和范围覆盖了由算法生成、后来又被销毁的所有构形Δ，这里的 j(Δ)表示销毁构形Δ的那轮迭代的编号。令 i(Δ)表示生成构形Δ的那轮迭代的编号。既然 $i(\Delta) \le j(\Delta) - 1$，就有：

$$\frac{n-[j(\Delta)-1]}{j(\Delta)-1} = \frac{n}{j(\Delta)-1} - 1 \le \frac{n}{i(\Delta)} - 1 = \frac{n-i(\Delta)}{i(\Delta)}$$

将其代入等式 (9.7)，就得到

$$\sum_{r=1}^{n} d \cdot \binom{n-r}{r} \cdot \mathrm{card}(\mathcal{T}(X_r)\backslash\mathcal{T}(X_{r+1})) \le \sum_{\Delta} d \cdot \binom{n-i(\Delta)}{i(\Delta)}$$

而该不等式的右端不会超过

$$\sum_{r=1}^{n} d \cdot \binom{n-r}{r} \cdot \mathrm{card}(\mathcal{T}(X_r)\backslash\mathcal{T}(X_{r-1}))$$

二者之间的差别，在于那些生成后没有被销毁的构形。

于是就有：

$$E[\sum_{r=1}^{n} \sum_{\Delta \in \mathcal{T}_r\backslash\mathcal{T}_{r-1}} \mathrm{card}(K(\Delta))] \le \sum_{r=1}^{n} d \cdot \binom{n-r}{r} \cdot E[\mathrm{card}(\mathcal{T}(X_r)\backslash\mathcal{T}(X_{r-1}))]$$

最后，根据〖定理 9.14〗，就可以得到我们所需要的一个上界：

$$E[\sum_{r=1}^{n} \sum_{\Delta \in \mathcal{T}_r\backslash\mathcal{T}_{r-1}} \mathrm{card}(K(\Delta))] \le \sum_{r=1}^{n} d \cdot \binom{n-r}{r} \cdot \frac{d}{r} \cdot E[\mathrm{card}(\mathcal{T}(X_r))]$$

❑

至此已在将条件抽象化之后，完成了对结构变化量的分析。如果你想试试，可以应用这一概括性的结论，来分析我们用以构造 Delaunay 三角剖分的那个随机增量式算法。特别地，可以证明：

$$\sum_{\Delta} \mathrm{card}(K(\Delta)) = O(n\log n)$$

这里的求和范围，覆盖该算法所生成的全部三角形Δ；而集合 K(Δ)则由落在该三角形外接圆内的所有点组成。

对于不是处于一般性位置的点集，我们也希望能够定义出一个构形空间，且其中的构形都是三角形。但不幸的是，这看起来似乎是不可能的。因此，在选取构形的时候，我们只好稍做调整。

设 X 为平面上的一个点集（它不见得处于一般性位置）。你应该记得，集合Ω中的元素，就是

在开始构造之初，我们所引入的三个附加点。在选取Ω中的这些点时，我们已经保证了它们不致于对联接于 P 中各点之间的 Delaunay 边造成任何的破坏。由 X ∪ Ω中不共线的三个点构成的任一三元组$\Delta = (p_i, p_j, p_k)$，都定义了一个满足 $D(\Delta) := \{p_i, p_j, p_k\}$的构形；而组成集合 K(Δ)的各点，要么落在三角形 $p_ip_jp_k$ 的外接圆的内部，要么落在该圆介于 p_i 和 p_k 之间、经过 p_j 的那段弧上。这样的一个构形Δ，称作 X 的一个 Delaunay 隅（Delaunay corner）——因为，Δ在 S ⊆ X 上是活跃的，当且仅当沿着 Delaunay 图 $\mathcal{DG}(\Omega \cup S)$中的某张面的边界，$p_i$、$p_j$ 和 p_k 为依次相邻的三个顶点。请注意，不共线的任何三点，都定义了三个互异的构形。

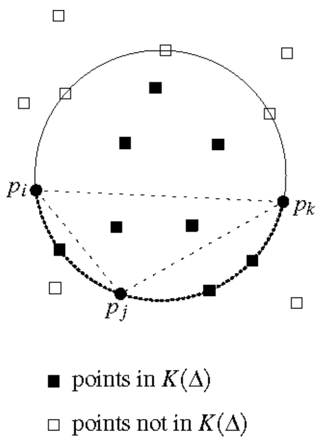


图9-24 K(Δ)由落在三角形$p_ip_rp_j$外接圆内的点组成（实心点为落在K(Δ)内的点；空心点为落在K(Δ)外的点）

一个重要的观察结论是：由算法DELAUNAYTRIANGULATION生成的每一个新三角形，都可以表示为$p_ip_rp_j$的形式——其中，p_r为本次迭代中将要插入的那个点；而$\overline{p_rp_i}$和$\overline{p_rp_j}$都是来自Delaunay图$\mathcal{DG}(\Omega \cup P_r)$中的边（请回顾〖引理 9.10〗）。由此可知，在生成三角形$p_ip_rp_j$的时候，三元组(p_i, p_r, p_j)就是$\mathcal{DG}(\Omega \cup P_r)$中的一个Delaunay隅；而且正因为如此，它必然是一个在集合P_r上活跃的构形。至于该构形所定义的集合K(Δ)，则是由落在三角形$p_ip_rp_j$外接圆内的那些点组成的（如 图 9-24 所示）。因此，可以将最初的那个求和式界定为

$$\sum_{\Delta} \mathrm{card}(K(\Delta))$$

任何一个 Delaunay 隅Δ，只要它曾经在构造过程的某一步中，出现在某个 Delaunay 图 $\mathcal{DG}(\Omega \cup P_r)$中，都属于上述求和式的范围。

现在可以利用〖定理 9.15〗。在S∪Ω对应的Delaunay图中，总共有多少个Delaunay隅呢？当该Delaunay图已是一个三角剖分时，即最坏的情况。若S由r个点组成，则该三角剖分必由 2(r+3) - 5 个三角形组成；相应地，共有 6(r+3) - 15 = 6r + 3 个Delaunay隅。于是，由〖定理 9.15〗可知：

$$\sum_{\Delta} \text{card}(K(\Delta)) \leq \sum_{r=1}^{n} 9 \cdot \left(\frac{n-r}{r}\right) \cdot \left(\frac{6r-3}{r}\right) \leq 54n \cdot \sum_{r=1}^{n} \frac{1}{r} \leq 54n(\ln n + 1)$$

至此，〖定理 9.12〗已经完全得证。

9.6 注释及评论

计算几何领域所研究的点集的三角剖分问题，即使在其它的领域中，也是广为人知的。二维或者更高维点集的三角剖分，对数值分析（比如有限元分析）以及图形学来说，都是极为重要的（一项预处理技术）。本章所讨论的三角剖分，仅限于一种情况——只能利用给定的点做为三角剖分的顶点。若允许使用附加的点——即所谓的Steiner点（steiner point）——则对应的问题就称为网格化（meshing），第 14 章将对此进行研究。

Lawson[244]曾经证明：通过（一系列的）边翻转操作，同一平面点集的不同三角剖分之间都可以相互转化。后来他还建议通过反复地进行边翻转，来找到一个好的三角剖分——每经过一次这样的边翻转，三角剖分的某个代价函数就会有一定的改善 [245]。

在早期的一段时间内，人们已经注意到，所谓插值效果好的三角剖分，应该尽量避免狭长三角形的出现 [38]。然而直到后来，才由Sibson[360]将这一性质归纳为：若不考虑退化情况，则以角度向量为标准，局部最优的三角剖分必是唯一的——而这个三角剖分，也就是Delaunay三角剖分。

这种方法只注意到了角度向量，而完全没有考虑到各数据点的具体高度，因此，它也被称作与数据无关的方法（data-independent approach）。Rippa[328]曾经给出过采用这种方法的理由，根据他的证明，无论具体的高度如何分布，在所有的三角剖分中，Delaunay三角剖分总能够使所生成地形的粗糙度（roughness）最小化。不过，根据他的定义，粗糙度等于地形梯度的L_2-模（L_2-norm）的平方的积分。后来的一些研究工作，试图通过引入高度信息这一因素来改进三角剖分。这类所谓的与数据相关的方法（data-dependent approach），是由Dyn等人 [154]首先提出的。根据数据点的高度，他们提出了对三角剖分进行评价的多种准则。有趣的是，他们也是从Delaunay三角剖分出发，然后通过一系列的边翻转操作，不断得到改进后的三角剖分。针对分片的三次插值，Quak和Schumaker[325]也采用了与此相同的方法；Brown[76]的工作，也是如此。Quak和Schumaker注意到，在逼近光滑表面方面，他们所得到的三角剖分虽然比Delaunay三角剖分有所改进，但是并不大；然而若换成不光滑的表面，效果就会有天壤之别。

Delaunay三角剖分与Voronoi图互为对偶，关于这个问题，第 7 章中给出了更多的参考文献。

本章所介绍的随机增量式算法，来自Guibas等人 [196]；不过，其中对$\sum_{\Delta}$ card(K(Δ))的分析方法，则是来自Mulmuley的专著 [290]。将该方法推广到退化点集的论证方法则是新的。Boissonnat等人[69][71]以及Clarkson与Shor[133]，还分别提出了其它一些随机算法（randomized algorithm）。

人们已经发现，由某一点集P得出的很多种几何图（geometric graph），都是P的Delaunay三角剖分的子图。其中最重要的一个几何图，恐怕就要数点集的欧几里得最小支撑树（Euclidean minimum spanning tree - EMST）[349]了。其它的还包括Gabriel图（Gabriel graph）[186]，以及相对邻近图（relative neighbor graph）[374]。习题部分将对这些几何图进行讨论。

还有一种重要的三角剖分，就是所谓的最小权三角剖分（minimum weight triangulation）[12][42][146][147]——亦即，以所采用各边的长度为权，总权重最小的那个三角剖分。在任一点集的所有三角剖分中确定最小权三角剖分，在最近已被 [291]证明是NP-完全的（NP complete）。

9.7 习题

习题 9.1 试讨论如下问题：由平面上任意 n 个点组成的一个集合，可能有多少个三角剖分?

a. 试证明：对于由 n 个点组成的集合，可能的三角剖分不会超过 $2^{\binom{n}{2}}$个；

b. 试证明：（对任何 n，总是）存在由 n 个点组成的一个集合，该集合至少有 $2^{n-2\sqrt{n}}$ 个可行的三角剖分。

习题 9.2 在三角剖分中，每个点的度数等于与之关联的边数。试构造出一个由 n 个点组成的平面点集，它必须具有这样的性质：无论如何对其进行三角剖分，其中总是存在一个点的度数为 n-1。

习题 9.3 试证明：对于同一平面点集的任何两个三角剖分，我们总是可以通过（一系列的）边翻转操作，将其中的一个三角剖分转换为另一个。提示：首先证明，给定同一凸多边的任何两个三角剖分，总是可以通过（一系列的）边翻转操作，将其中的一个三角剖分转换为另一个。

习题 9.4 试证明：对于所有顶点都共圆的任何一个凸多边形（convex polygon），任何三角剖分的最小角都是相等的。这就意味着，点集的任何一个 Delaunay 三角剖分，都将使最小角达到最大。

习题 9.5 a. 试证明：对于平面上的任意四个点 p、q、r 和 s，其中 s 落在 p、q 和 r 的外接圆之内，当且仅当如下条件成立（假定点 p、q 和 r 按逆时针方向构成一个三角形）：

$$\det\begin{pmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{pmatrix} > 0$$

b. 只要对上面的行列式进行测试，就可以判断三角剖分中的一条边是否合法。除此之外，你能否给出其它的方法来实现这一判断？将你的方法与上面的方法进行比较，然后讨论二者各自的优、缺点。

习题 9.6 算法 DELAUNAYTRIANGULATION 调用了一个递归过程 LEGALIZEEDGE。试将该过程改写成迭代的形式，并将其与递归的版本做一比较，然后讨论二者各自的优、缺点。

习题 9.7 试证明：出现在$\mathcal{DG}(P_r)$中却不属于$\mathcal{DG}(P_r)$的任何一条边，必与p_r相关联。也就是说，正如图 9-18 所示的那样，在$\mathcal{DG}(P_r)$中出现的各条新边，必然构成一个（以p_r为中心的）星形结构。不借助于算法DELAUNAYTRIANGULATION，试对此给出一个直接的证明。

习题 9.8 设n个点构成一个处于一般性位置的集合P，$q \notin P$为P的凸包内的任一点。在P的Delaunay三角剖分中，令$p_ip_jp_k$为包含q的一个三角形。（注意，由于q可能正好落在Delaunay三角剖分中的某条边上，故这样的三角形可能有两个[①]。）试证明：$\overline{qp_i}$、$\overline{qp_j}$和$\overline{qp_k}$必然都是$P \cup \{q\}$的Delaunay三角剖分中的边。

习题 9.9 本章所给出了一个随机算法，它能够在 $O(n\log n)$的期望时间内，构造出任意 n 个点的 Delaunay 三角剖分。试证明：在最坏情况下，该算法的运行时间为$\Omega(n^2)$。

习题 9.10 按本章所介绍的算法，在开始构造Delaunay三角剖分之前，首先要引入两个附加点p_{-1}和p_{-2}。它们不能落在任何三个输入点的外接圆内，而且可以按照字典序看到P中各点。为了保证这一点，这里的做法是：在实现涉及它们的操作时，须做特殊处理（参见第 9.3 节）。如果企图避免做如此特殊处理，需要显式地计算出这两个点的具体坐标。请你给出计算的方法。（相对于本章介绍的方法，）这种方法更好吗？

习题 9.11 对于任一平面点集 P，其欧几里得最小支撑树（Euclidean minimum spanning tree - EMST）被定义为“联接 P 中所有点、长度最短的树”。在很多应用中，都需要通过通讯线路（比如构建局域网）、公路、铁路等诸如此类的形式，将平面环境中的多个基点联结起来。这类问题都将涉及到 EMST。

[①] 又，由于 $q \notin P$，故这样的三角形至多只有两个。——译者

a. 试证明：在P的Delaunay三角剖分的边集中，包含了P的一棵EMST[①]；

b. 试利用上述性质设计一个算法，在 $O(n\log n)$ 时间内构造出 P 的一棵 EMST。

习题 9.12 所谓的旅行商问题（traveling salesman problem - TSP），就是在给定若干个点之后，计算出一条遍历（traverse）所有点的最短路径。旅行商问题是 NP-难的（NP hard）。试借助上题所定义的 EMST，设计一个 TSP 的近似算法（approximate algorithm）。该算法所生成的遍历路径，长度不超过最优解的两倍。

习题 9.13 任意平面点集P的Gabriel图（Gabriel graph）定义如下：其中两点p和q为该图贡献一条边，当且仅当以pq为直径的圆的内部没有P中其它任何点（如 图 9-25 所示）。

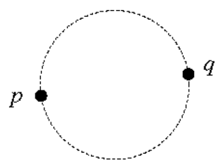


图9-25 pq为Gabriel图的一条边，当且仅当以pq为直径的圆内部为空

a. 试证明：P 的 Gabriel 图是 $\mathcal{DG}$(P)的一个子图；

b. 试证明：在 P 的 Gabriel 图中，p 和 q 为相邻的两点，当且仅当由 p 和 q 确定的那条 Delaunay 边，与和它对偶的 Voronoi 边相交；

c. 试给出一个算法，在 $O(n\log n)$ 时间内，构造出任意 n 个点的 Gabriel 图。

习题 9.14 对任意的平面点集 P，其相对邻近图（relative neighborhood graph - RNG）定义如下：两个点 p 和 q 为相对邻近图贡献一条边，当且仅当

$$d(p, q) \leq \min_{r \in P\setminus\{p,q\}} \max(d(p, r), d(q, r))$$

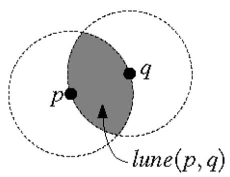


图9-26 pq为相对邻近图的一条边，当且仅当lune(p,q)内部为空

对于任意两点 p 和 q，令 lune(p,q)为分别以 p 和 q 为中心、以 d(p, q)为半径的两个圆的公共部分。

a. 试证明：p 和 q 为相对邻近图贡献一条边，当且仅当 lune(p, q)的内部不含 P 中的任何点；

[①] 注意：同一点集可能有多棵 EMST。——译者

b. 试证明：P 的相对邻近图为 $\mathcal{DG}$(P)的一个子图；

c. 试给出一个算法，构造出任意点集的相对邻近图。

习题 9.15 试证明：对于任意点集 P，其 EMST、RNG、GG 以及 $\mathcal{DG}$ 的边集具有如下关系：

$$\text{EMST} \subseteq \text{RNG} \subseteq \text{GG} \subseteq \mathcal{DG}$$

（其中各图的定义，参见以上的几道习题。）

习题 9.16 对于由任意n个点组成的集合P，所谓P的一个k-聚类（k-clustering），就是将P划分为非空的子集$P_1, ..., P_k$[①]。对于其中任意两个子集P_i和P_j，它们之间的距离被定义为P_i中各点到P_j中各点的最短距离，亦即：

$$\text{dist}(P_i, P_j) := \min_{p \in P_i, q \in P_j} \text{dist}(p, q)$$

我们希望（对任意的 P 和 k）找出一个 k-聚类，使得各子集之间的最短距离达到最大。

a. 假设子集（P_i和 P_j）之间的最短距离是由 $p \in P_i$和 $q \in P_j$实现的。试证明：$\overline{pq}$必为 P 的 Delaunay 三角剖分中的一条边。

b. 试给出一个算法，在 $O(n\mathbf{log}n)$时间内构造出一个 k-聚类，使得其中各子集之间的最短距离达到最大。提示：使用并查数据结构（union-find data structure）。

习题 9.17 所谓三角剖分的权重（weight），就是其中所有边的总长度。构造最小权三角剖分（minimum weight triangulation），就是在任一点集的所有三角剖分之中，找出权重最小者。 有人曾猜测：Delaunay 三角剖分就是最小权三角剖分。请举一反例。

习题 9.18* 试构造一个几何的构形空间(X, Π, D, K)，其中的 $\mathcal{T}(X_r)\backslash\mathcal{T}(X_{r+1})$与 $\mathcal{T}(X_{r+1})\backslash\mathcal{T}(X_r)$相比，可以大到任意程度。

习题 9.19* 利用构形空间（的概念及结论），对第 6 章中的随机增量式算法重新做一分析。

[①] $P_i \cap P_j = \varnothing$，$1 \le i < j \le k$。——译者

10

更多几何数据结构：截窗

将来，多数汽车上都会配备一套车辆导航系统，它可以帮助你确定你的位置，并引导你通往目的地。这样一个系统中存有一张路线图，比如美国的公路图。系统也会跟踪你的位置，这样，无论何时，你总能在车载计算机的屏幕上看到自己所处局部的路线图。这样的局部范围，通常是你当前所在位置周围的一块矩形区域。有的时候，该系统甚至还能为你提供更多的帮助。比如，它可能会提醒你，必须在前方的某个岔路口拐弯，否则就无法到达目的地。

为发挥作用，该地图必须提供充分的信息细节。整个欧洲的一份详细地图中，含有极大量的数

据。幸运的是，地图中只有一小部分内容需要显示。当然，系统还是应该能够在地图中找出这些部分：给定一个矩形区域——称为截窗（window）——系统必须能够确定，地图中的那些部分（公路、城市等）落在截窗内部，并显示这些内容。这个过程，称为截窗查询（windowing query）。

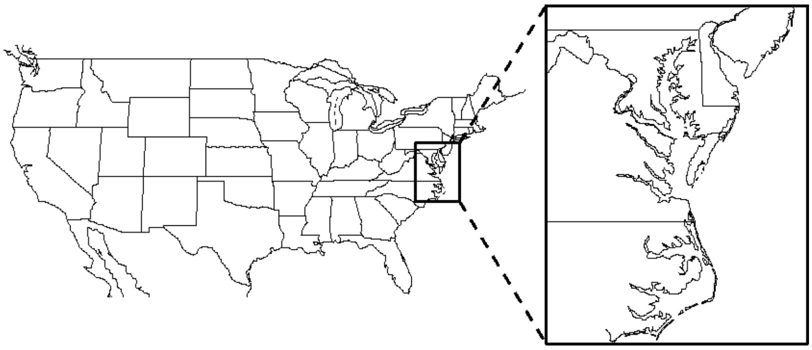


图10-1 在美国地图上的一次截窗查询

对地图的各个部分逐一检查，看看它是否落在截窗的内部，是一种直截了当的方法。然而鉴于此时所处理数据的规模之大，这并不是一个可行的方法。我们的方法是，通过将地图存储为某种数据结构，使得我们可以快速地抽取出落在特定截窗之内的部分。

截窗查询不仅对地理图的操作有用。在其它（如计算机图形学或 CAD/CAM 等）一些应用领域中，截窗查询也扮演着重要的角色。其中的一个例子，就是飞行模拟。组成地形模型的三角面片，数量可能极大，但是在飞行员的视野范围之内，通常只是整个地形的很少一部分。因此，只能选取位于某一给定区域之内的那部分地形。这一问题中的区域是三维的，故被称作视体（view volume）。

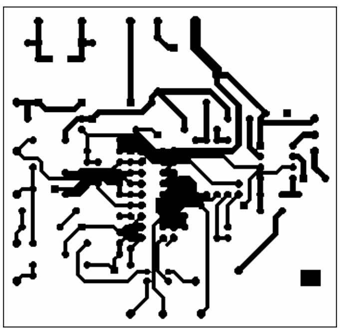


图10-2 印刷电路板对应于平面图画

另一个例子来自印刷电路板的设计（参见第 14 章）。如 图 10-2 所示，这类设计所得到的结果，一般包括（若干层）由线路及元件构成的平面图画（planar drawing）。在设计过程中，设计员时常需要放大电路板上某一特定的部分，以便更为仔细地进行观察。同样地，我们在这里所需要做的，就是在电路板上所有的线路和元件当中，找出落在截窗之内的那些。实际上，无论何时，只要人们需要对一个巨大而复杂对象的某一很小局部进行查看，就需要用到截窗技术。

第 5 章所介绍的区域查找与截窗查询非常类似。二者的差别，在于各自所处理数据的类型——前者所处理的数据是点集，而后者所处理的数据通常都是线段、多边形或曲线之类。另外，我们往往会在高维查找空间中进行区域查找，而截窗查询的查找空间通常都仅限于二维或三维。

10.1 区间树

我们首先从上面所给例子中最容易的那个入手——这就是对一块印刷电路板的截窗。这一问题之所以比其它问题更加容易，是因为其处理的数据是受限制的——印刷电路板上的物体，边界通常都是由线段组成的，而且这些线段的方向只有少数几种可能。它们常常会与电路板的侧边平行，或者与某条侧边成 45 度角。这里只考虑所有线段都与某一条侧边平行的情况。换而言之，如果认为x-轴与电路板的底边重合，y-轴与左边重合，那么每条线段都与x-轴或y-轴平行——如 图 10-3 所示，我们称它们是与坐标轴平行的（axis-parallel）或正交的（orthogonal）。同时假定，查询窗口（query window）也是与坐标轴平行的——即它是一个各边都与坐标轴平行的矩形。

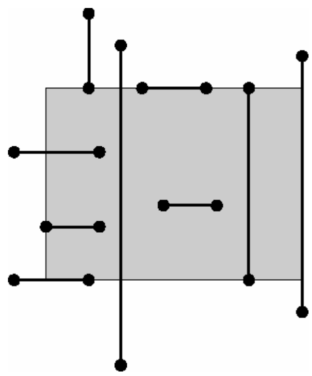


图10-3 正交截窗查询

设S为一组共n条与坐标轴平行的线段。为了解答截窗查询，需要使用一种数据结构来存放S，使得对于任何给定的查询窗口W := [x : x'] × [y : y']，与W相交的所有线段总可以有效地被报告出来。首先来看看，线段可能会如何与截窗相交。可以分为以下情况：线段可能完全落在W内部，可能与W的边界有一个交点，也可能有两个交点，也可能与W的边界（部分）叠合（即有无数个交点）。在多数情况下，线段至少有一个端点落在W内。只要将S中各线段所有的 2n个端点组成一个集合，并针对截窗W进行一次区域查找，就可以找出所有这类线段。在第 5 章中，我们曾经见过能够支持这一操作的一种数据结构——区域树。一棵二维的区域树需要占用$O(n\log n)$空间，而且每次区域查找都可以在$O(\log^2 n + k)$时间内完成，其中k为实际被报告出来的点数。我们也曾经证明过：借助分散层叠技术，可以将查询时间改进到$O(\log n + k)$。这种方法有一个小问题。如果我们真地利用W对由所有

线段端点构成的点集进行区域查找，那些两个端点都落在W中的线段就会被报告两次。这个问题不难解决——在每条线段第一次被找出来之后，我们立即对该线段做个记号，只有那些未做记号的线段，才需要报告出来。另一种解决方法是，一旦在W中找到了某条线段的一个端点，我们将随即检查该线段的另一个端点，看看它是否也落在W内。若没有落在W内部，则报告该线段。要是另一个端点也落在W之内，那么只有在当前端点为（所属线段的）左端点或下端点的时候，才报告该线段。上述分析可以归纳为如下引理：

〖引理 10.1〗

设 S 为平面上一组共 n 条与坐标轴平行的线段。对于任一与坐标轴平行的查询窗口，我们都可在 $O(\mathbf{log}n + k)$时间内，将至少有一个端点落在该查询窗口内的所有线段报告出来；为此，需要使用一个占用 $O(n\mathbf{log}n)$空间的数据结构，其预处理时间为 $O(n\mathbf{log}n)$。其中，k 为实际被报告出来的线段条数。

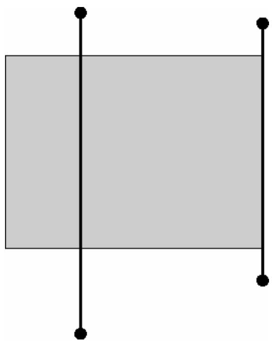


图10-4 与查询窗口相交、两个端点都落在查询窗口之外的线段

下面要讨论的问题是，如何找出两个端点都落在查询窗口之外（但仍然与查询窗口相交）的那些线段（如 图 10-4 所示）。这些线段或者与W的边界有两个交点，或者与W边界上的某条边（部分）叠合。如果是一条垂直线段，它将与边界上的两条水平边相交。如果是水平线段，就会与两条垂直边相交。因此，只要报告出与W边界的左边或顶边相交的所有线段，就可以（进一步从中）找出所需的线段。（请注意，并不需要对边界上的另外两条边进行查询。）更准确地说，我们只需要报告出两个端点都落在W之外的那些线段——因为，其它的线段在此前已经被报告出来了。不妨集中考虑这样一个问题：找出与W的左侧边相交的所有水平线段。我们只需交换x-和y-坐标，就可以按照同样的方法处理顶边。

我们面对的是这样一个问题：对平面上的一组水平线段S进行预处理，使得对于任一垂直查询线段，我们都能够有效地将S中与之相交的所有线段报告出来。为了加深对该问题的理解，首先来看看一个简单的版本：如 图 10-5 所示，查询“线段”是一条完整的直线。设$l := (x = q_x)$为这条查询直线。水平线段$s := (x, y)(x', y)$与l相交，当且仅当$x \le q_x \le x'$。因此，在这种条件下，只有线段的x-坐标

才起作用。换而言之，这个问题已退化为这样一个一维的问题：在实轴上给定一组区间，要求将包含待查询点（query point）q_x的所有区间报告出来。

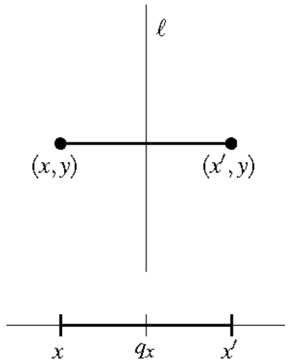


图10-5 问题简化：针对直线的查询

设 $I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$为实轴上的一组闭合区间。为了同时考虑到原来二维的问题，我们把该实轴假想为水平的，而且将“小（大）于…”说成是“位于…的左（右）边”。令 x_{mid} 为这 2n 个区间端点的中值（median）。于是，在 x_{mid} 的左边和右边，至多各有一半的区间端点。如果查询值 q_x 落在 x_{mid} 的左边，那么显然，完全落在 x_{mid} 右边的区间都不可能包含 q_x。根据这种构思，可以构造出一棵二叉树。在该树的右子树中存放的，是由完全落在 x_{mid} 右边的区间所组成的一个集合 I_{right}；在左子树中存放的，是由完全落在 x_{mid} 左边的区间所组成的一个集合 I_{left}。对左、右两棵子树，按照同样的方法进行递归构造。这里有一个问题需要处理：对于包含 q_x 的那些区间，又将如何处理呢？一种可能的方法是，将这些区间同时存放在左、右两棵子树中。

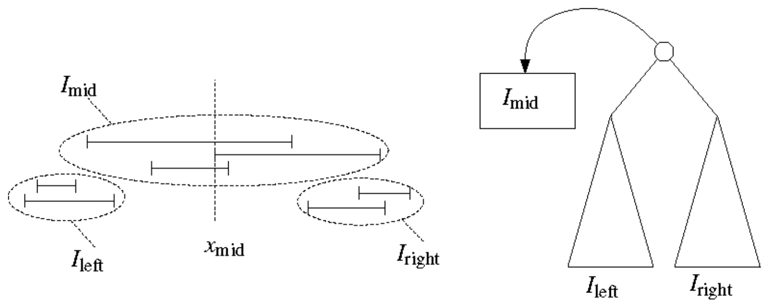


图10-6 根据x_{mid}，对所有线段进行分类

然而这样一来，对于节点的孩子，这一问题依然可能再次发生。于是，一个区间将可能被多次存储，从而导致该数据结构占用的空间过于庞大。为了克服区间重复存储的问题，可以采用另外一种方法：对于由包含x_{mid}的区间所组成的集合I_{mid}，我们将其单独存储为另一个结构，并将该结构指定给树的根节点。具体方法参见 图 10-6。请注意，在这幅（以及其它的）图中，尽管所有区间实际上都落在实轴上，我们还是将它们画在不同的高度上，以示区别。

借助这些新的关联结构，即可将I_{mid}中包含q_x的所有区间报告出来。至此又回到了此前所提出的

问题：给定一组区间I_{mid}，要求从中找出包含q_x的所有区间。然而，如果运气不佳，I_{mid}可能会与I完全一样，如 图 10-7 所示。

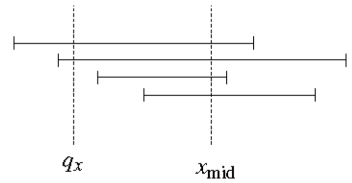


图10-7 运气不佳时，I_{mid}可能与I完全一样

表面看来，似乎我们又转回了原地，但实际上已经有所区别。现在我们已经知道了，I_{mid} 中的每个区间都包含了 x_{mid}——这一点十分有用。比如，假设 q_x 位于 x_{mid} 的左侧。在这种情况下，我们就可以推断出，I_{mid} 中每一区间的右端点必然位于 q_x 的右侧。因此，只有各区间的左端点才是重要的——q_x 落在区间$[x_j : x_j']\in I_{mid}$ 内，当且仅当 $x_j \le q_x$。如果按照左端点递增的次序，将各区间组织成一个有序表，那么只要 q_x 落在某个区间内，它就必然同时落在该区间在列表中的每一个前趋区间之内。也就是说，可以沿着列表（从前往后）顺序扫描各个区间，一旦发现某个区间不包含 q_x，即可立即停止。此时之所以可以停止，是因为后面的区间绝不可能包含 q_x。类似地，若 q_x 位于 x_{mid} 的右侧，也可以沿着一个存放右端点的列表，（从前往后）顺序扫描各个区间。此时的这个列表，必须按照右端点坐标递减的次序来组织——只有这样，当待查询点 q_x 落在 x_{mid} 右侧时，才能对其进行遍历（traversal）。最后，还有一种可能是 $q_x = x_{mid}$。此时，只需将 I_{mid} 中的所有区间报告出来。（我们并不需要专门处理这种情况。实际上，只要顺序扫描其中任一有序表即可。）

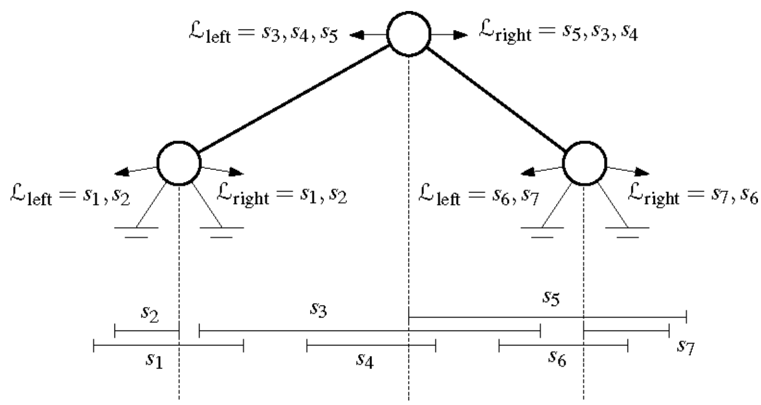


图10-8 区间树

接下来，对用以存放I中各区间的整个数据结构做一简要描述。该数据结构被称为区间树（interval tree）。图 10-8 就是一棵区间树，其中的垂直虚线，分别表示各节点所对应的x_{mid}值。

- 若 $I = \emptyset$，则区间树为一匹叶子。
- 否则，取 x_{mid} 为所有区间端点的中值。令

$I_{left} := \{[x_j : x_j'] \in I \mid x_j' < x_{mid}\}$

$I_{mid} := \{[x_j : x_j'] \in I \mid x_j \leq x_{mid} \leq x_j'\}$

$I_{right} := \{[x_j : x_j'] \in I \mid x_{mid} < x_j\}$

其对应的区间树的根节点为 v，x_{mid} 就存放在 v 处。此外，

- 集合 I_{mid} 被存储了两遍——一次存放在按照各区间左端点排序的列表 $\mathcal{L}_{left}(v)$中，另一次存放在按照各区间右端点排序的列表 $\mathcal{L}_{right}(v)$中；
- v 的左子树为与子集 I_{left} 对应的一棵区间树；
- v 的右子树为与子集 I_{right} 对应的一棵区间树。

〖引理 10.2〗

任意 n 个区间所对应的区间树，占用 $O(n)$空间，深度为 $O(\log n)$。

〖证明〗

深度的上界（upper bound）不证自明，故只需证明存储空间的上界。我们注意到，子集 I_{left}、I_{mid} 和 I_{right} 互不相交。这样，每一区间只能在一个集合 I_{mid} 中出现一次，于是，每一区间只能在两个有序表中各出现一次。这就说明，所有关联列表所占用的空间总量不会超过 $O(n)$。另外，树本身也只占用 $O(n)$的存储空间。 ❑

根据区间树的定义，可以直接得到一个递归构造区间树的算法，如下所示。（你应该记得，lc(v)和 rc(v)分别表示节点 v 的左、右孩子。）

算法 ConstructIntervalTree(I)

输入：实轴上的一组区间 I

输出：集合 I 所对应区间树的根节点

1. **if** (I = ∅)
2.   **then return**(一匹空叶子)
3.   **else** 生成一个节点 v
计算全部区间端点的中值 x_{mid}，并将 x_{mid} 存入 v 中
4. 计算出 I_{mid}，根据 I_{mid} 构造出两个有序表：
所有左端点的有序表 $\mathcal{L}_{left}(v)$
所有右端点的有序表 $\mathcal{L}_{right}(v)$
并将这两个列表存入 v 中
5. $lc(v) \leftarrow$ ConstructIntervalTree(I_{left})
6. $rc(v) \leftarrow$ ConstructIntervalTree(I_{right})

7. **return** v

确定一组点的中值，只需要线性的时间。尽管如此，与第 5 章的情况一样，更好的办法反而是首先对所有点进行一次排序，并找出其中的中值。这样，在随后的各次递归调用中，就可以很容易对这些经过预排序的集合进行更新。设$n_{mid} := card(I_{mid})$，则生成列表$\mathcal{L}_{left}(v)$和$\mathcal{L}_{right}(v)$需要$O(n_{mid}\mathbf{log}n_{mid})$时间。因此，我们所需要的时间（不计递归调用所需的时间）为$O(n + n_{mid}\mathbf{log}n_{mid})$。与〖引理 10.2〗的证明方法同理，我们可以证明：该算法的运行时间为$O(n\mathbf{log}n)$。

〖引理 10.3〗

一组共 n 个区间所对应的区间树，可以在 $O(n\mathbf{log}n)$时间内构造出来。

还有一个问题需要解释：借助区间树，如何才能找出包含 q_x 的所有区间？我们已经就此做过扼要的描述，下面就来给出具体的算法过程。

算法 QUERYINTERVALTREE(v, q_x)

输入：以 v 为根节点的一棵区间树，以及一个待查询点 q_x

输出：包含 q_x 的所有区间

1. **if** (v 不是叶子)
2. **then if** ($q_x < x_{mid}(v)$)
3. **then** 遍历列表 $\mathcal{L}_{left}(v)$：
   从其中最左侧端点所属的区间开始，逐一报告出包含 q_x 的各区间
   一旦到达不包含 q_x 的（第）一个区间，即终止遍历
4. QUERYINTERVALTREE(lc(v), q_x)
5. **else** 遍历列表 $\mathcal{L}_{right}(v)$：
   从其中最右侧端点所属的区间开始，逐一报告出包含 q_x 的各区间
   一旦到达不包含 q_x 的（第）一个区间，即终止遍历
6. QUERYINTERVALTREE(rc(v), q_x)

上述算法的查询时间，不难分析。每访问一个节点 v，只需要花费 $O(1 + k_v)$时间，其中 k_v 为在 v 处所报告出来的区间数目。所有被访问过的节点所对应的 k_v 累加起来，自然应该等于 k。此外，在树中的任一深度，我们最多访问一个节点。正如前面已经提到的，区间树的深度为 $O(\log n)$。因此，总的查询时间就是 $O(\log n + k)$。

上述有关区间树的结果，可以归纳为如下定理：

〖定理 10.4〗

给定一组共 n 个区间 I，其对应的区间树占用 $O(n)$空间，而且能够在 $O(n\log n)$时间内构造出来。对于任一待查询点，都可以借助区间树，在 $O(\log n + k)$时间内将包含该点的所有区间报告出来，其中 k 为实际被报告出来的区间数目。

至此可以稍事停顿并回味一下，上述结果到底对我们有何帮助。我们最初所要解决的问题是：将一组与坐标轴平行的线段组织成某种数据结构，以使得对于任一指定的截窗$W = [x : x'] \times [y : y']$，都能够（有效地）找出与W相交的所有区间。使用第 5 章所介绍的数据结构——区域树——可以找出（至少）有一个端点落在W之内的所有线段。与W相交的其它线段，必然与W的边界相交两次[①]。为了找出这些线段，我们打算分别使用W的左侧边和底边[②]进行查询。这样，就需要某种数据结构来存放一组水平线段，使得与任何垂直的待查询线段相交的所有线段，可以被有效地报告出来；类似地，还需要一个数据结构来存放一组垂直线段，以支持针对任何水平线段的求交查询。我们从一个稍微简单些的问题入手，并建立了一种数据结构来解决这一问题——这个问题之所以简单些，是因为其中的待查询对象是一条完整的直线。由此引出了区间树。现在，就来看看，应该如何对区间树做推广，使之支持针对垂直线段的查询（如 图 10-9 所示）。

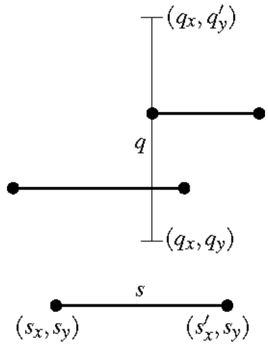


图10-9 针对垂直线段的查询

令$S_H \subseteq S$为S中所有水平线段组成的子集，设q为一条垂直的待查询线段$q_x \times [q_y : q'_y]$。对于S_H中的任何一条线段$s := [s_x : s'_x] \times s_y$，我们将$[s_x : s'_x]$称为该线段的x-区间（x-interval）。假定已经按照x-区间的次序，将S_H中的所有区间组织成了一棵区间树$\mathcal{T}$。让我们对查询算法QUERYINTERVALTREE实际检查一遍，看看在针对某一垂直的待查询线段q对$\mathcal{T}$进行查找的过程中，会出现什么情况。以存储于树$\mathcal{T}$根节点处的x_{mid}值为界，假定q_x落在左侧。这种情况下，有一点依然成立：我们只需要对左子树进

① 更准确地说，应该是“至少有两个交点”。——译者

② 原著第二版误作“top edge”，虽勘误已指出应为“bottom edge”，但可惜第三版仍未予更正。——译者

行递归查询。之所以能够跳过右子树，是因为完全落在x_{mid}右侧的那些线段不可能与q相交。然而，原先对集合I_{mid}的处理方式在此却不能继续套用。现在，线段$s \in I_{mid}$的左端点落在q的左侧，还不足以说明它必然与q相交；另一个必要条件是，s的y-坐标必须落在$[q_y : q_y']$之间。这一点可以从 图 10-10 中看出。

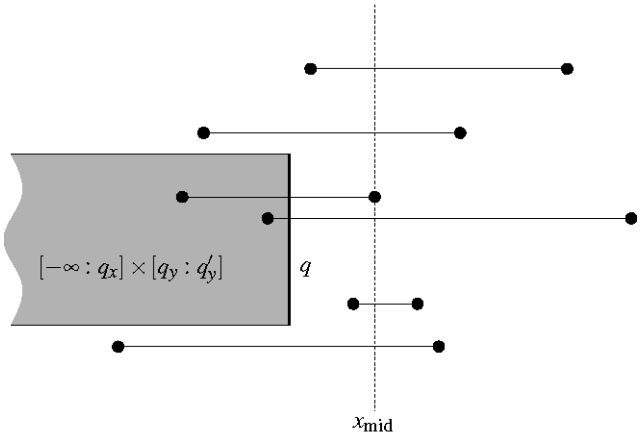


图10-10　与q相交的每一条线段，其左端点必然落在阴影区域内

因此，仅仅将所有端点组织成一个有序表，还是不够的。为此，需要另一种更为精致的关联结构：对于任一待查询区域$(-\infty : q_x'] \times [q_y : q_y']$，我们都能利用这一结构，将左端点落在该区域中的所有线段报告出来。反之，若q落在x_{mid}的右侧，也应该能够将其右端点落在区域$[q_x : +\infty) \times [q_y : q_y']$中的所有线段报告出来——为了处理这种情况，还需要另一个关联结构。那么，如何实现这种关联结构呢？实际上，我们所需要进行的查询，不折不扣地就是针对某一（平面）点集的矩形区域查询。于是顺理成章地，使用第 5 章所介绍的二维区域树结构就行了。按照这种方法，关联结构将占用$O(n_{mid}\log n_{mid})$的存储空间，其中$n_{mid} := card(I_{mid})$，而对应的查询时间则为$O(\log n_{mid} + k)$。

这样，用以存放 S_H 中各水平线段的数据结构，形式如下。其主结构为存放所有线段 x-区间的一棵区间树 $\mathcal{T}$。在每个节点 v 处，我们存放的不再是两个有序表 $\mathcal{L}_{left}(v)$和 $\mathcal{L}_{right}(v)$，而是如下的两棵区域树：前一棵区域树 $\mathcal{T}_{left}(v)$存放的是 $I_{mid}(v)$中所有线段的左端点；后一棵区域树 $\mathcal{T}_{right}(v)$存放的是 $I_{mid}(v)$中所有线段的右端点。就其占用的存储量而言，区域树要比有序表高出一个 $\log n$ 的因子，因此该数据结构总共占用的存储量将达到 $O(n\log n)$。预处理时间依然是 $O(n\log n)$。

现在的查询算法与 QUERYINTERVALTREE 基本相同，二者只有一点差别——在这里，不再是对有序表 $\mathcal{L}_{left}(v)$进行遍历（traversal），而是对区域树 $\mathcal{T}_{left}(v)$进行查询。因此在每个节点上，只需要花费 $O(\log n + k_v)$时间，其中 k_v 为（在 v 处）实际被报告出来的线段条数。既然查找路径由不超过 $O(\log n)$ 个节点组成，故总的查询时间就变成了 $O(\log^2 n + k)$。

由此可得如下定理：

〖定理 10.5〗

设 S 为平面上的一组共 n 条水平线段。对于任一垂直待查询线段（query segment），我们都可在 $O(\mathbf{log}^2 n + k)$时间内，将 S 中与该线段相交的所有线段报告出来，其中 k 为实际被报告出来的线段条数；为此需要借助一个数据结构，该结构占用 $O(n\mathbf{log}n)$空间，并可在 $O(n\mathbf{log}n)$时间内被构造出来。

该结论与〖引理 10.1〗综合起来即可得到一种方法，解决针对与坐标轴平行的线段的截窗问题。

〖推论 10.6〗

设 S 为平面上一组共 n 条与坐标轴平行的线段。对于任一与坐标轴平行的矩形查询窗口，我们都可在 $O(\mathbf{log}^2 n + k)$时间内，将 S 中与该截窗相交的所有线段报告出来，其中 k 为实际被报告出来的线段条数；为此需要借助一个数据结构，该结构占用 $O(n\mathbf{log}n)$空间，并可在 $O(n\mathbf{log}n)$时间内被构造出来。

10.2 优先查找树

第 10.1 节介绍了支持截窗的数据结构，其中的关联结构是通过区域树来实现的。针对这种结构进行的区域查找，具有如下特殊性质：每次查询都在某一侧无界。本节将介绍另一种数据结构，称为优先查找树（priority search tree），这种结构能够利用上述特殊性质，将存储性能改进至$O(n)$。这一结构同时也简单许多，因为它并不需要进行分散层叠。利用优先查找树而不是区域树这一数据结构来解决截窗问题，可以将〖定理 10.5〗中的存储上界改进至$O(n)$。不过，〖推论 10.6〗中的存储上界并不会因此有所改进——因为，为了报告出落在截窗中的所有端点，仍然需要使用一棵区域树。

设$P := \{p_1, \ldots, p_n\}$为平面上的一组点。我们希望设计出一种结构，来求解形如$(-\infty : q_x] \times [q_y : q_y']$的矩形查询。为了理解如何对这种特殊性质加以利用，让我们来看看一维的情况。一般的一维区域查找，就是要找出落在某个区域$[q_x' : q_x]$之内的所有点。为了能够有效地找出这些点，可以如第 5 章所介绍的那样，将所有的点组织成一棵一维的区域树。如果待查询区域的左侧是无界的，实际上就是要查找出落在$(-\infty : q_x]$内的所有点。我们可以更高效地解决这个问题，方法是：从最左侧的点开始对一个有序表进行遍历（traversal），一旦遇到第一个没有落在该区域中的点就立即终止遍历。采用这种方法，查询时间为$O(1 + k)$，而不再是通常情况所需要的$O(\mathbf{log}n + k)$。

现在，又该如何将这一策略推广至左侧无界的二维区域查找呢？无论如何，如果不使用关联结构，就必须将 y-坐标的信息集成到该结构中来，这样才能在 x-坐标落在$(-\infty : q_x]$内的所有点中，轻松地找出 y-坐标落在$[q_y : q_y']$内的那些点。要是采用一个简单的线性列表，并不足以漂亮地完成这一任务。因此，我们采用了另一种结构——堆（heap）。

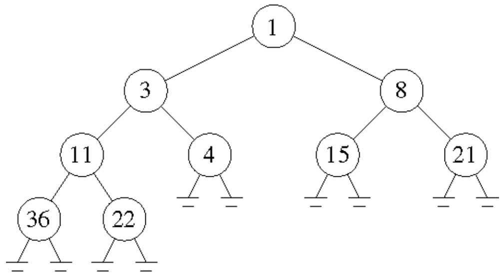


图10-11 集合{1, 3, 4, 8, 11, 15, 21, 22, 36}对应的一个堆

我们通常都是利用堆来支持优先级查询（priority query）——即从一组数值中找出最小（或最大）者。不过，堆也可以用以支持形如$(-\infty : q_x]$的一维区域查找。就查询时间而言，堆与有序表相同，都是$O(1 + k)$。通常，堆优于有序表的方面在于：无论是插入新的点，还是将最大的点删除，都可以更加高效地完成。对我们来说，堆的树形结构还有另一个优点：它可以使得对 y-坐标的集成更加容易——我们马上就会看到这一点。

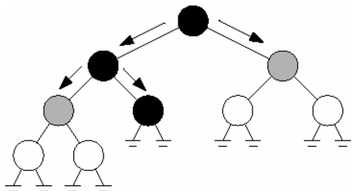


图10-12 堆

如 图 10-12 所示，所谓的堆，就是定义如下的一棵二叉树。该树的根节点存放了最小的x-值。集合中其余的点被分成规模接近的两个子集；这两个子集也是按照同样的规则，分别递归地存储。如 图 10-11 所示的，就是堆的一个例子。我们只要从上而下对这棵树搜索一趟，就可以回答一次针对$(-\infty : q_x]$的查询。每搜索到一个节点，都要检查一下存储于该节点处的点，看看它的x-坐标是否落在$(-\infty : q_x]$内。如果是，就报告出该点，并对其左、右子树继续搜索下去；否则，就终止对树中该部分的搜索。例如，在针对区间$(-\infty : 5]$对 图 10-11 中的那棵树进行搜索时，将报告出点 1、3 和 4。我们也会访问到节点 8 和 11，不过一旦到达这些位置后，就不再会从那里继续搜索下去。

得益于堆的灵活性，每个集合都可以任意地划分为两个子集。如果还希望对y-坐标进行查找，就可以采用一个技巧——在划分子集的时候，不是像普通的堆那样随意进行划分，而是根据y-坐标来进行划分。更准确地说，在将（除根节点以外的）其余各点划分为两个子集的时候，我们不仅要使子集的规模相互接近，而且前一子集中每个点的y-坐标，都要小于后一子集中的所有点。这一点如 图 10-13 所示。在该图中，树被画成了朝向侧面的，其目的是为了说明此处的划分是按照y-坐标进行的。在 图 10-13 的示例中，点p_5的x-坐标最小，所以存放在根节点中。其余的（五个）点按照y-坐标一分为二。其中，点p_3、p_4和p_6的y-坐标更小，所以归入左子树。在这三个点中，p_3的x-坐标最小，于是该点就存放在左子树的根节点处。依此类推。

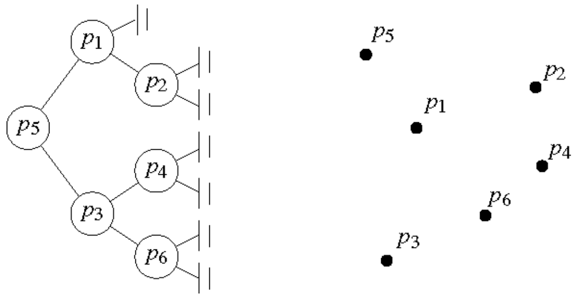


图10-13　一个点集及其对应的优先查找树

对于任一点集P，其优先查找树的形式化定义如下。我们假定，所有点的坐标互异。在第 5 章（更准确地说，是第 5.5 节）中我们已经了解到，这一假定条件并不会降低该方法的一般性；只要采用合成数之类的方法，就可以模拟出“所有点坐标均互异 [①]”这一条件。

- 若 $P = \emptyset$，则对应的优先查找树为一匹空的叶子。
- 否则，设 p_{min} 为集合 P 中 x-坐标最小的那个点。令 y_{mid} 为其余诸点 y-坐标的中值。取

  $P_{below} := \{p \in P \backslash \{p_{min}\} \mid p_y < y_{mid}\}$

  $P_{above} := \{p \in P \backslash \{p_{min}\} \mid p_y > y_{mid}\}$

  对应的优先查找树的根节点为 v，其中存放了点 $p(v) := p_{min}$，以及数值 $y(v) := y_{mid}$。此外，

  - v 的左子树为对应于集合 P_{below} 的一棵优先查找树；
  - v 的右子树为对应于集合 P_{above} 的一棵优先查找树。

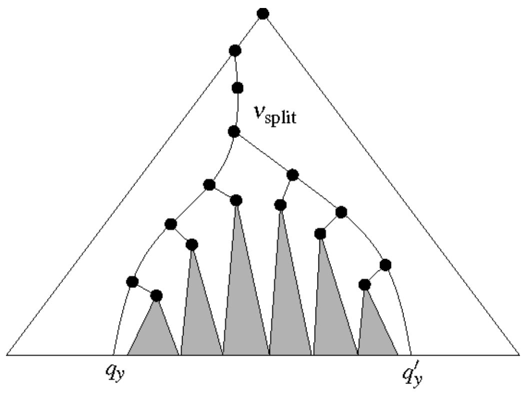


图10-14　对一棵优先查找树进行查询

由此，可以直接导出一个构造优先查找树的 $O(n\log n)$算法。有趣的是，只要所有点已经按照 y-坐标排好了序，其对应的优先查找树居然可以在线性时间内构造出来。此方法的思想是，自下而上（而不是自上而下）地进行构造，具体的构造过程与普通堆相同。

① 更为准确的表述是“各点的 x-坐标互异，各点的 y-坐标互异”。——译者

针对区域$(-\infty : q_x] \times [q_y : q_y']$对一棵优先查找树的查询过程，大致如下。首先，分别针对q_y和q_y'进行查询，其结果如 图 10-14 所示。其中所有用阴影表示的子树，只包含y-坐标落在指定区域内的点。因此接下来只需按照x-坐标，逐一检查这些子树即可。这项工作可以由下面的子程序完成，它基本上就是堆的查询算法。

```
算法 REPORTINSUBTREE(v, q_x)
输入：以 v 为根节点的一棵优先查找树，以及数值 q_x
输出：该子树中 x-坐标不超过 q_x 的所有点
1.    if (v 不是一匹叶子，而且(p(v))_x ≤ q_x)
2.        then 报告 p(v)
3.             REPORTINSUBTREE(lc(v), q_x)
4.             REPORTINSUBTREE(rc(v), q_x)
```

〖引理 10.7〗

REPORTINSUBTREE(v, q_x)可以在$O(1 + k_v)$时间内，从以 v 为根节点的子树中，将 x-坐标不超过 q_x 的所有点报告出来，其中 k_v 为实际被报告出来的点数。

〖证明〗

在以 v 为根节点的子树中，任取满足下述条件的一个节点μ：μ中所存放的点 p(μ)满足$(p(\mu))_x \le q_x$。根据该数据结构的定义，沿着从μ通往 v 的路径，各节点所存储的点的 x-坐标构成一个递减序列，即这些点的 x-坐标均不超过q_x。因此，在其中每一节点处，搜索都不会终止——由此可知，（搜索）必然会到达μ，而 p(μ)也必然会被报告出来。于是可以得出结论：x-坐标不超过 q_x 的每一个点都会被报告出来。反过来，显然只有这些点才会被报告出来。

在访问到的每一个节点 v 处，只需要消耗 $O(1)$时间。一旦访问到一个满足$\mu \ne v$ 的节点μ，我们必然已经在μ的父节点处报告过另一个点。我们将消耗在μ处的时间记到这个点的“账”上。这样一来，每个被报告出来的点都记了两次账——这就说明，如果$\mu \ne v$，我们消耗在节点μ处的时间就是$O(k_v)$。再加上消耗在 v 处的时间，总共消耗的时间就是$O(1 + k_v)$。 ❑

如果对每一棵被挑拣出来的子树（即 图 10-14 中用阴影表示的那些子树），都调用一次 REPORTINSUBTREE，就可以找出所有落在待查询区域中的点吗？答案是否定的。例如，既然树的根节点所存储的是x-坐标（全局）最小的点，它很可能就落在待查询区域内。实际上。在（从根节点）到q_y和q_y'的两条查找路径上，每一节点所存储的点都有可能落在待查询区域之内——因此，我们还需要对这些节点进行检查。由此可得如下查询算法：

算法 QUERYPRIOSEARCHTREE($\mathcal{T}$, $(-\infty : q_x] \times [q_y : q'_y]$)

输入：一棵优先查找树$\mathcal{T}$，以及一个左侧无界的待查询区域

输出：落在该区域内的所有点

1. 在$\mathcal{T}$中分别查找 q_y 和 q'_y
   设 v_{split} 为两条查找路径分道扬镳处的那个节点
2. **for** (q_y 和 q'_y所对应查找路径上的每一个节点 v)
3. **do if** ($p(v) \in (-\infty : q_x] \times [q_y : q'_y]$) **then** 报告 p(v)
4. **for** (在 v_{split} 的左子树中、在 q_y 所对应查找路径上的每一个节点 v)
5. **do if** (该查找路径在 v 处拐向左侧)
6. **then** REPORTINSUBTREE(rc(v), q_x)
7. **for** (在 v_{split} 的右子树中、在 q'_y所对应查找路径上的每一个节点 v)
8. **do if** (该查找路径在 v 处拐向右侧)
9. **then** REPORTINSUBTREE(lc(v), q_x)

〖引理 10.8〗

算法 QUERYPRIOSEARCHTREE 可以在 $O(\log n + k)$时间内，报告出落在待查询区域$(-\infty : q_x] \times [q_y : q'_y]$内的所有点，其中 k 为实际被报告出来的点数。

〖证明〗

首先证明：由该算法报告出来的每一个点，都落在待查询区域之内。对于（从根节点到）q_y和q'_y的搜索路径上的每个（节点所存储的）点，这一点不言而喻——因为这些点必然会经过显式的测试，从而判断出它们是否落在待查询区域内。试考察算法第 6 行中对 REPORTINSUBTREE(rc(v), q_x)的某次调用。设p为在调用过程中被报告出来的一个点。根据〖引理 10.7〗，可得$p_x \le q'_x$。另外，由于这次调用过程中访问到的所有节点都居于v_{split}的左侧，而且$q'_y > y(v_{split})$，故有$p_y \le q'_y$。最后，由于这次调用过程中访问到的所有节点都居于v的右侧，而且针对q_y的搜索路径在v处朝左拐（left-turn），故有$p_y \ge q_y$。对于在第 9 行中被报告出来的那些点，同理可证。

以上证明了：所有被报告出来的点都落在待查询区域内。反过来，设 p(μ)为该区域中的任一点。在针对 q_y 的搜索路径朝右拐（right-turn）的任何一个节点处，其左子树中所有点的 y-坐标都应小于 q_y。同理可以说明：在针对 q'_y的搜索路径朝左拐的任何一个节点处，其右子树中所有点的 y-坐标都应大于 q'_y。因此，μ要么属于两条搜索路径之一，要么属于某棵被调用了 REPORTINSUBTREE 的子树。无论是何种情形，p(μ)都会被报告出来。

最后来分析该算法的运行时间。它应该线性正比于分布在针对q_y和q_y'的搜索路径沿途之上的节点数目，再加上对子程序REPORTINSUBTREE的所有调用所需的时间。这棵树的深度为$O(\log n)$，故每条搜索路径由$O(\log n)$个节点组成。根据〖引理 10.7〗，对REPORTINSUBTREE的所有调用需要$O(\log n + k)$时间。❑

优先查找树的性能，可以归纳为如下定理：

〖定理 10.9〗

对于由平面上任意 n 个点组成的集合 P，我们都可在$O(n\log n)$时间内为其构造出一棵占用$O(n)$存储空间的优先查找树。借助这棵优先查找树，对于任何形如$(-\infty : q_x] \times [q_y : q_y']$的待查询区域，我们都可在$O(\log n + k)$时间内从 P 中找出落在该区域中的所有点，其中 k 为实际被报告出来的点数。

10.3 线段树

至此，我们已经对针对一组与坐标轴平行的线段的截窗问题进行了讨论。为解决这一问题，我们建立了一种很好的数据结构，该结构使用了区间树，并以优先查找树作为其联合结构。之所以将处理的对象限制于与坐标轴平行的线段，是着眼于解决诸如印刷电路板设计等应用问题。然而，如果查询的对象换成路线图，这一限制条件就不再满足了——此时，线段的方向显然可以是任意的。

通过一种技巧，可将一般性的区域查找问题，转换为限制于与坐标轴平行的线段的一个问题。

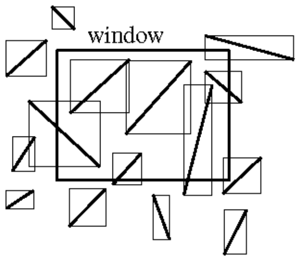


图10-15　利用包围框（bounding box），将一般性查询转换为正交查询

为此，需要将每条线段替换成它的包围框（如 图 10-15 所示）。这样，使用此前针对与坐标轴平行的线段所建立的那种数据结构，就可以找出与查询窗口W相交的所有包围框。接下来，只要逐一检查与W相交的各包围框中的每条线段，就可以判断它们本身是否与W相交[①]。在实践中，这一技术的效果通常还不错——在其包围框与W相交的那些线段中，绝大部分线段本身也会与W相交。然而

[①] 实际上，这一条件并不充分。比如，将 W 的对角线朝两端分别延长ε > 0，这样得到的一条线段 s 与 W 相交，但组成 s 包围框的四条线段与 W 都不相交。——译者

在如 图 10-16 所示的最坏情况下，这种方法的效率极差——在这种情况下，每一个包围框都与W相交，而任何线段都不与W相交。因此，为了确保查询速度足够快，必须求助于其它的方法。

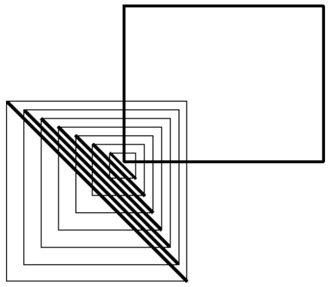


图10-16 最坏情况

与此前的做法一样，我们也将与W相交的所有线段划分为两类：至少有一个端点落在W内的线段，以及与截窗边界相交的线段 ①。借助区域树，可以报告出所有的第一类线段。为了找出第二类线段，我们需要针对截窗的四段边界，分别做一次相交查询。（当然，这里需要仔细处理，以保证每条线段最多只被报告一次。）我们将只以垂直边界为例，说明进行查询的方法。至于水平边界，方法是类似的。这样，就可以假定输入S为平面上的一组方向任意的线段，而我们的目标则是从S中找出与某一垂直待查询线段$q := q_x \times [q_y : q_y']$相交的所有线段。我们还假定S中的线段互不相交，但允许（在端点处）首尾衔接。（要是允许线段相交，问题的难度就会增加很多，而且时间复杂度的上界也会很高。如果的确需要处理这种情况，可以求助于第 16 章将要介绍的技术。）

首先来看看，是否可以直接采用前一节所介绍的方法，来处理方向任意的一组线段。通过在区间树中搜索q_x，我们可以挑选出若干个子集$I_{mid}(v)$。对于任一被选出来的节点v，只要$x_{mid}(v) > q_x$，$I_{mid}(v)$中每一条线段的右端点都必然会落在q的右侧。如果这是一条水平线段，那么它与待查询线段相交的充要条件就是，它的左端点落在区域$(-\infty : q_x] \times [q_y : q_y']$内。然而，如果允许线段取任意的方向，情况就不再那么简单了——如 图 10-17 所示，即使已经知道线段的右端点落在q的右侧，仍然于事无补 ②。因此，区间树并不适用于这种情况。下面将试图构造出另一种可以解决一维问题的数据结构，而且这种结构更加适宜于处理方向任意的线段。

① 此处对第二类线段的定义不够准确。改为“…，以及与截窗边界至少有两个交点的线段”似更妥。——译者

② 具体地讲，对于右端点落在 q 之右侧的一条线段，其左端点落在区域$(-\infty : q_x] \times [q_y : q_y']$之内，并不意味着该线段与待查询线段相交；反之，即使其左端点落在区域$(-\infty : q_x] \times [q_y : q_y']$之外，该线段也不见得与待查询线段不相交。此处的插图只反映了前一种情况，请读者自己给出后一情况的实例。——译者

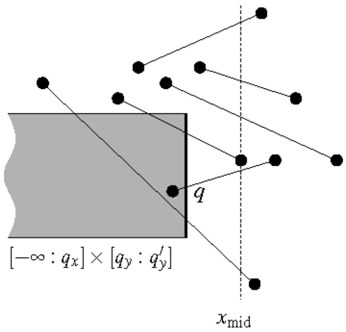


图10-17 线段方向任意时，截窗查询无法分解为沿两个正交方向的两次查询

设计数据结构的模式之一，就是所谓的轨迹法（locus approach）。每一次查询，都可以用若干个参数来描述。以截窗问题为例，需要四个参数：q_x、q'_x、q_y 和 q'_y。对任何一组参数的组合，我们能得到某一具体的回答。相近的参数组合，常常会对应于同一个答案；如果我们将截窗稍微移动一点，与之相交的往往还是原先的那些线段。我们将参数所有可能的组合合在一起，称之为参数空间（parameter space）。就截窗问题而言，其对应的参数空间是四维的。按照轨迹法，就是要将参数空间划分成多个子区域，使得在同一子区域内的任何查询都对应于同一个答案。这样，无论是什么查询，只要能够确定它（在参数空间中）所属的子区域，我们也就可以得到所需的答案。只有在划分出来的子区域数目不大的情况下，这种方法才能奏效。然而就截窗问题而言，这一条件并不成立。事实上，子区域的数目可以高达$\Theta(n^4)$。尽管如此，按照轨迹法，我们还是能够得出一种新的数据结构，以替换区间树。

设$I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$为实轴上的一组共n个区间。我们所希望找到的数据结构，应该能够对任一待查询点q_x，报告出S中包含q_x的所有区间。这种查询只需一个参数q_x描述，因此其对应的参数空间为一条实轴。将各区间的端点从左到右排成序列$p_1, \ldots, p_m$（其中剔除了重合的端点）。所谓对参数空间的划分，只不过是在这些点p_i的位置对实轴进行分割。如 图 10-18，如此分割所得到的子区域，称作基本区间（elementary interval）。划分出来的基本区间从左到右依次是：

$$(\infty : p_1), [p_1 : p_1], (p_1 : p_2), [p_2 : p_2], \ldots, (p_{m-1} : p_m), [p_m : p_m], (p_m : +\infty)$$

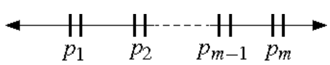


图10-18 基本区间

在这个基本区间的序列中，开区间与闭区间交替出现：每一个开区间，都介于某两个相邻端点 p_i 和 p_{i+1} 之间；而每一个闭区间，都是某一端点本身。之所以要将所有点 p_i 本身都看作一个区间，自然是因为在一个区间的内部的查询结果，与在区间端点处不见得完全相同。

为找出包含待查询点 q_x 的那些区间，必须找出包含 q_x 的那些基本区间。为此，需要建立一棵二分查找树 $\mathcal{T}$，树中的叶子分别对应于各基本区间。与某匹叶子μ对应的基本区间，记作 $Int(\mu)$。

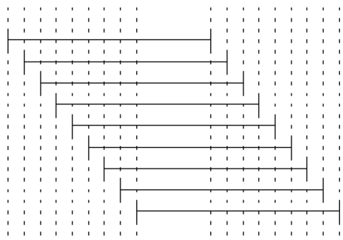


图10-19 最坏情况下，每个区间都被重复地存放线性次

如果将I中包含Int(μ)的所有区间（经排序后）存放在叶子μ处，就可以在$O(\log n + k)$时间内，报告出包含q_x的k段区间——首先，用$O(\log n)$时间在T中搜索q_x；然后，用$O(1 + k)$时间报告出存放在μ处的所有区间。照此方法，可以有效地回答查询。问题是，如此一来，该数据结构将需要占用多大的存储量呢？跨越很多段基本区间的那些区间，将被重复地存储在这种数据结构的多匹叶子中。因此，要是各区间之间（部分）叠合的现象很普遍，总的存储量将会很大。如 图 10-19 所示，倘若我们运气不佳，存储量甚至会高达平方量级。下面就来看看，是否有办法降低其存储消耗量。

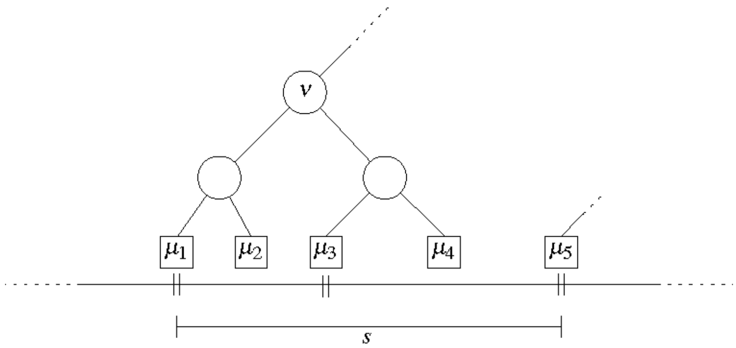


图10-20 将线段s存放在v处，而不是分别存放在μ_1、μ_2、μ_3和μ_4处

如 图 10-20 所示，有一个区间跨越了五个基本区间。考察分别与基本区间μ_1、μ_2、μ_3和μ_4相对应的四匹叶子。如果针对q_x的搜索路径终止在其中的某匹叶子处，我们就必须报告这段区间。关键的一个观察结果是：搜索路径终止于μ_1、μ_2、μ_3或μ_4，当且仅当该路径穿过内部节点v。

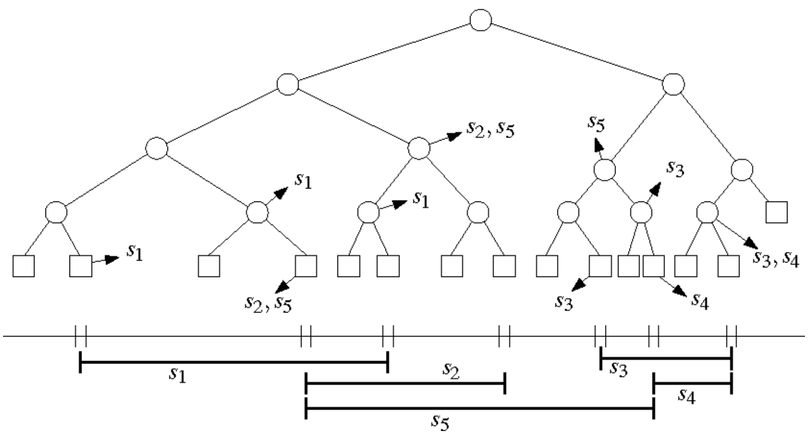


图10-21 线段树：其中的节点通过箭头，指向对应的正则子集

既然如此，为什么还要将该段区间分别存放在叶子μ_1、μ_2、μ_3 和μ_4（以及μ_5 处）处，而不干脆将它存放在节点v处（以及μ_5 处）呢？一般而言，每段区间都分别被存放在若干个节点处，这些节点的并集首先必须覆盖这段区间，而且我们会尽可能地选用层次更高的节点。基于这一原理的数据结构，称为线段树（segment tree）。下面将针对一组区间I所对应的线段树，给出精确的描述。如 图 10-21 所示的，就是与五段区间相对应的一棵线段树。

- 线段树的主体结构，为一棵平衡二分查找树 $\mathcal{T}$。I 中的所有区间导出了一组基本区间，而 $\mathcal{T}$ 中的叶子就按照某种次序，分别对应于这些基本区间——最左侧的叶子对应于最左端的基本区间，其余依此类推。与叶子μ相对应的基本区间，记作 Int(μ)。
- $\mathcal{T}$ 中的每个内部节点，都对应于由（至少两段）基本区间合并而成的某段区间——与节点 v 相对应的区间 Int(v)，就是在以 v 为根节点的子树中，所有叶子μ对应的基本区间 Int(μ)的并集。（这同时说明，Int(v)也是它的两个孩子各自对应区间的并集。）
- 在 $\mathcal{T}$ 中的任一节点或叶子 v 处，都存放了一段区间 Int(v)，以及区间的一个集合 $I(v) \subseteq I$（可以采用链表的形式来组织）。这个集合，就是对应于节点 v 的正则子集，它由 I 中的一些区间$[x : x'] \in I$ 组成，这些区间必须满足：$Int(v) \subseteq [x : x']$，而且 $Int(parent(v)) \nsubseteq [x : x']$。

下面就来看看，将区间尽可能存放于高层节点中的这种策略，是否的确有助于存储量的降低。

〖引理 10.10〗

n 段区间所对应的线段树只占用 $O(n\log n)$空间。

〖证明〗

既然 $\mathcal{T}$ 是一棵由不超过 4n + 1 匹叶子组成的平衡二分查找树，其高度应为 $O(\log n)$。我们断言：在 $\mathcal{T}$ 的任一深度，每个区间$[x : x'] \in I$ 最多记录于两个节点（所对应的集合 I(v)）中。

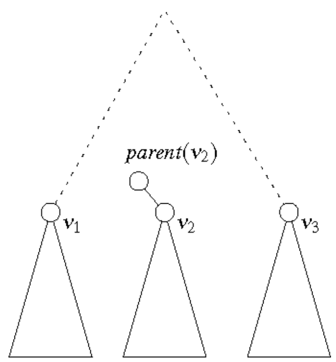


图10-22 同一深度上的三个节点v_1、v_2和v3

为证明这一断言，任取三个深度相同的节点v_1、v_2和v3（从左向右编号，如 图 10-22 所示）。

假设[x : x']存储于v_1和v_3处。这就意味着，[x : x']跨越了从Int(v_1)的左端点到Int(v_3)的右端点的整段区间。因为v_2处于v_1和v_3之间，所以Int(parent(v_2))必然也包含在[x : x']内。于是，[x : x']就不可能同时存放在v_2中。由此可以得出结论：在$\mathcal{T}$的任一指定深度上，每段区间都只能存放在不超过两个节点中——因此，总的存储量为$O(n\log n)$。 ❑

由此看来，上述策略的确行之有效——我们已经将最坏情况下的存储量，从平方量级降低到了$O(n\log n)$。不过另一方面，查询（时间）又会有何变化？依然能够轻松地完成查询吗？答案是肯定的。具体的过程，可以描述为下面这个简单的算法。该算法的首次调用，使用的（参数）是 v = root($\mathcal{T}$)。

算法 QUERYSEGMENTTREE(v, q_x)

输入：线段树（或者其某棵子树）的根节点，以及待查询点 q_x

输出：树中包含 q_x 的所有区间

```
1.  报告 I(v)中的所有区间
2.  if (v 不是叶子)
3.      then if (qx ∈ Int(lc(v))
4.          then QUERYSEGMENTTREE(lc(v), qx)
5.          else QUERYSEGMENTTREE(rc(v), qx)
```

在树中的每一层，该查询算法只访问一个节点，因此总共需要访问 $O(\log n)$个节点。在每个节点 v 处，花费的时间都是 $O(1 + k_v)$，其中 k_v 为实际被报告出来的区间数目。由此，可以得出如下引理：

〖引理 10.11〗

借助线段树，可以在 $O(\log n + k)$时间内，报告出包含待查询点 q_x 的所有区间，其中 k 为实际被报告出来的区间数目。

可按照下面的方法，构造一棵线段树。首先，需要花费 $O(n\log n)$时间，对 I 中所有区间的端点进行排序。由此可以得到一组基本区间。接下来，需要将所有基本区间组织成一棵平衡二分查找树；对于树中的每一个节点 v，都要确定其代表的区间 Int(v)。按照自底而上的次序，可以在线性时间内完成这项任务。最后，需要计算出每个节点各自对应的正则子集。为此，可以将各区间逐一插入到线段树中。为了将一段区间插入到$\mathcal{T}$中，可以使用参数 v = root($\mathcal{T}$)来调用下面的子程序：

算法 INSERTSEGMENTTREE(v, [x : x'])

输入：线段树（或者其某棵子树）的根节点，以及一段区间

输出：该区间被插入到（子）树中

```
1.  if (Int(v) ⊆ [x : x'])
2.      then 将[x : x']存放到 v 中
```

3.	**else** **if** (Int(lc(v) ∩ [x : x'] ≠ ∅)
4.	**then** INSERTSEGMENTTREE(lc(v), [x : x'])
5.	**if** (Int(rc(v) ∩ [x : x'] ≠ ∅)
6.	**then** INSERTSEGMENTTREE(rc(v), [x : x'])

为了将一段区间[x : x']插入到线段树中，需要花费多少时间呢？在被访问到的每一个节点处，我们只需要花费常数时间（假定 I(v)都是以诸如链表之类的简单结构存储的）。在访问每个节点 v 的时候，我们要么将[x : x']存放在 v 处，要么 Int(v)中含有[x : x']的一个端点。我们已经知道，在$\mathcal{T}$中的同一层上，任何区间最多只能被存放两遍。另外，在每一层上，最多只能各有一个节点所对应的区间包含 x 或 x'。因此，在每一层上最多只需要访问四个节点。于是，插入单段区间所需的时间应为 $O(\log n)$，而构造线段树的总体时间为 $O(n\log n)$。

线段树的性能，可以归纳为如下定理。

〖定理 10.12〗

对于由任意 n 段区间组成的集合 I，我们都可在 $O(n\log n)$时间内，相应地构造出一棵占用 $O(n\log n)$空间的线段树。借助这棵线段树，对于任一待查询点，我们都可在 $O(\log n + k)$时间内，从 I 中找出包含该待查询点的所有区间，其中 k 为实际被报告出来的区间数目。

你应该还记得，区间树只占用线性规模的空间，而且利用这种数据结构，我们同样可以在 $O(\log n + k)$时间内，报告出包含指定待查询点的所有区间。因此如果只是限于这类任务，区间树就已经足矣。只有在需要回答更为复杂的查询（比如针对一组线段的截窗查询）时，诸如线段树之类更为强大的结构才会有用武之地。其原因在于，包含 q_x 的所有区间合起来，正好就是我们在对线段树进行搜索的过程中，所挑选出来的那些正则子集的并集。要是换成一棵区间树，尽管在每次查询的过程中我们也会挑选出 $O(\log n)$个节点，但这些节点中所存放的区间并不都会包含待查询点。这样，接下来还要通过对有序表进行遍历（traversal），才能最终找出相交的所有区间。而对线段树而言，我们则可以将所有的正则子集存放到对应的联合结构中，以支持更进一步的查询。

现在回到最初的截窗问题。设S为平面上方向任意、互不相交的一组线段。我们希望从S中找出与一段待查询线段$q := q_x \times [q_y : q_y']$相交的所有线段。我们来看看，如果按照x-区间将S中的各线段组织成一棵线段树$\mathcal{T}$，将会有何效果。这样，$\mathcal{T}$中的每个节点v都可以被看作是对应于一条垂直条带（slab）：$\mathrm{Int}(v) \times (-\infty : +\infty)$。如果一条线段完全横跨于v所对应的条带之上——此时我们称该线段跨越（span）该条带——同时又没有跨越v的父节点所对应的条带，那么它就属于v所对应的正则子集。该子集记

作S(v)。图 10-23 对此做了说明。

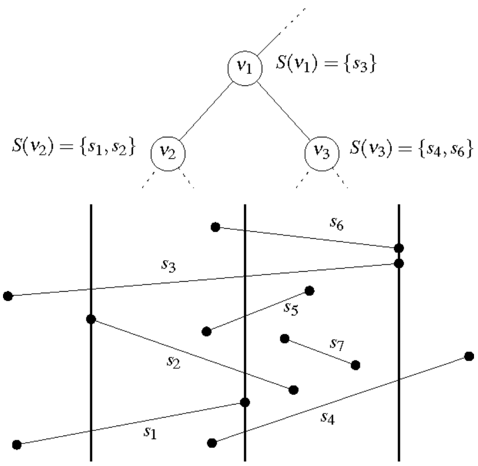


图10-23　正则子集中的各条线段，跨越了该节点所对应的条带，但不跨越其父节点所对应的条带

如果我们针对 q_x 对 $\mathcal{T}$ 进行搜索，就会得到 $O(\log n)$个正则子集——它们就是搜索路径沿途上各节点所对应的正则子集——它们合起来囊括了所有 x-区间包含 q_x 的那些线段。在这样一个正则子集中，任一线段 s 与 q 相交，当且仅当 q 的下端点比 s 更低，同时 q 的上端点比 s 更高。

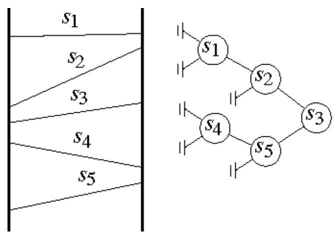


图10-24　沿垂直方向，与正则子集S(v)对应的查找树$\mathcal{T}$(v)

那么，如何才能将介于q的两个端点之间的所有线段找出来呢？为此需要利用这样一个事实：正则子集S(v)中的所有线段，必然都跨越节点v所对应的条带，而且这些线段互不相交。这就意味着，可以沿着垂直方向对这些线段进行排序。于是，如 图 10-24 所示，可以将S(v)存储为一棵按照垂直次序组织的查找树$\mathcal{T}$(v)。只要对$\mathcal{T}$(v)进行搜索，就能够在$O(\log n + k_v)$时间内，找出所有相交的线段，其中k_v为实际相交的线段数目。因此，存储集合S的主体数据结构如下：

- 集合 S 被存储为一棵按照各线段 x-区间组织的线段树 $\mathcal{T}$。
- $\mathcal{T}$ 中任一节点 v 所对应正则子集中的线段，均跨越 v 所对应的条带，但不跨越 v 的父节点所对应的条带。按照各线段在条带内的垂直次序，该正则子集存放于二分查找树 $\mathcal{T}$(v)中。

任何一个节点 v 的联合结构所占用的存储量，线性正比于 S(v)的规模，因此总共占用的存储量

为 $O(n\log n)$。这些联合结构可以在 $O(n\log n)$时间内被构造出来，因此总的预处理时间为 $O(n^2\log n)$。稍做进一步的努力，就可以将这一指标改进到 $O(n\log n)$。改进的思想是，在构造线段树的过程中，在各线段之间沿着垂直方向维护一个（偏）序。只要存在这样一个顺序，就可以在线性的时间内构造出所有的联合结构。

查询算法十分简单：按照通常的方法在线段树中搜索q_x；沿着搜索路径每到达一个节点v，就要在$\mathcal{T}(v)$中搜索q的上端点和下端点，从S(v)中找出与q相交的所有线段。这基本上就是一次一维的区域查找（参见第 5.1 节）。在$\mathcal{T}(v)$中的每次搜索需要$O(\log n + k_v)$时间，其中k_v为在v处实际报告出来的线段条数。于是，总的查询时间就是$O(\log^2 n + k)$，由此可以得出如下定理：

〖定理 10.13〗

设 S 为平面上一组共 n 条互不相交的线段。对于任一垂直待查询线段，我们都可在 $O(\log^2 n + k)$时间内，报告出 S 中与该线段相交的所有线段，其中 k 为实际被报告出来的线段条数。为此，我们需要借助于一个占用 $O(n\log n)$空间的数据结构，该结构可在 $O(n\log n)$时间内构造出来。

实际上，只要各线段不相交于内部即可。很容易验证，当允许线段的端点相互重合时，这种方法依然适用。由此可以得出如下推论：

〖推论 10.14〗

设 S 为平面上一组共 n 条内部互不相交线段。对于任一与坐标轴平行的矩形查询窗口，都可在 $O(\log^2 n + k)$时间内，从 S 中找出与其该窗口相交的所有线段，其中 k 为实际报告出来的线段条数。为此，需要借助于一个占用 $O(n\log n)$空间的数据结构，该结构可在 $O(n\log n)$时间内构造出来。

10.4 注释及评论

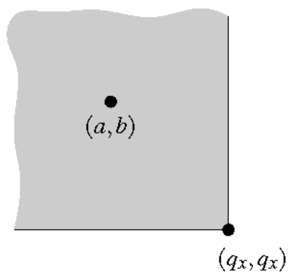


图10-25 区间[a : b]对应于点(a, b)

“找出包含某一给定点的所有区间”之类的查询，常常也被称作穿刺查询（stabbing query）。为解决这类问题，可以使用区间树结构，这种结构是由Edelsbrunner[157]和McCreight[270]提出的。

优先查找树则是由McCreight[271]提出的。McCreight同时注意到，优先查找树结构也可以用以解决穿刺查询问题。为此，只需要进行一个简单的转换——如 图 10-25 所示，将每个区间[a : b]映射到平面上的点(a, b)。这样，为了完成针对q_x的穿刺查询，只要进行一次针对区域$(-\infty : q_x] \times [q_x : +\infty)$的区域查找即可。而这种类型的区域，属于优先查找树能够支持的一种特殊情况。

线段树是由Bentley[45]首先提出的。不过，如果仅仅是将它当做一种一维的数据结构来解决穿刺查询问题，效率反而不如区间树——这是因为，它需要占用$O(n\log n)$空间。线段树的重要性主要在于，可以按照某种方式，将存放于各节点处的区间子集进行结构化组织，使得该问题能够方便地得到解决。因此，针对二维甚至更高维对象的处理，人们对线段树还做了很多种扩展与推广[211][157][163][301][375]。线段树比区间树更为强大的另一方面在于，只要对线段树结构做些调整，就可以很容易地解决所谓的穿刺计数查询（stabbing counting query）问题——即，统计包含某一给定待查询点的区间总数。针对这一问题，并不需要将各区间以列表的形式存放到每个节点中，只要存入一个代表具体数目的整数即可。对任一待查询点，只要把搜索路径上沿途各节点中存放的整数累加起来，就可完成查询。用来解决穿刺计数查询问题的这样一棵线段树，只需占用线性规模的空间，而查询时间为$O(\log n)$——因此，已经达到最优了。

曾经有些人研究过区间树及线段树的动态性（dynamization）问题——也就是说，使之能够插入和（或）删除区间。实际上，优先查找树最初的版本，就是一种完全动态的数据结构，为此要将普通的二分查找树，换成红黑树（red-black tree）[199][137]或者其它种类的平衡二分查找树——对这些结构的每次更新，只要进行$O(1)$次旋转。在很多环境中，输入都是不断变化的，动态性在这些场合就很重要。在许多平面扫描算法（plane sweep algorithm）中，通常都需要将状态结构组织为某种动态结构（dynamic structure），在这种情况下，动态数据结构也是很重要的。在解决一些问题时，也需要用到线段树的动态版本。

可分解搜索问题（decomposable searching problem）概念的提出，极大地促进了一大类数据结构朝动态性的发展 [46][48][166][254][269][276][304][306][307][308][337]。设S为在搜索问题中涉及到的一组对象，设$A \cup B$为S的一个划分。对于任何搜索问题，如果一旦获得了该问题分别针对A和B的答案，就总可以在常数的时间内导出该问题针对S的答案，那么就称这个问题是可分解的（decomposable）。比如，穿刺查询问题就是可分解的——无论将输入划分为哪两个子集，只要分别找出这两个子集中的穿刺区间，实际上也就报告出了整个集合中的穿刺区间。与此类似，穿刺计数查询也是可分解的——只要将两个子集各自对应的数目相加，就可得到整个集合对应的总数。还有一些搜索问题（比如区域查找问题 ）也是可分解的。只要问题是可分解的，就可以通过一种通用的方法，将原来（解决该问题）的静态数据结构转化为动态数据结构。有关这方面的综合介绍，请参阅Overmars的专著 [299]。

穿刺查询问题可以推广至高维空间。这种情况下，给定的是一组与坐标轴平行的（超）矩形，要求从中找出包含每一待查询点的所有（超）矩形。为了回答这种高维穿刺查询，可以使用一种多

层线段树（multi-level segment tree）结构。这种数据结构占用$O(n\log^{d-1}n)$空间，利用它可以在$O(\log^d n)$时间内回答每次查询。采用分散层叠技术（参见第 5 章），可以将这一上界降低一个对数因子。如果通过区间树来实现联合结构中的最低一层，还可以将其存储量降低一个对数因子。无论是区间树还是优先查找树，都不存在高维的版本——也就是说，不能通过简明的方式对这些结构进行推广，使之能够用以解决高维空间中类似的查询问题。不过，这些结构还是可以用来实现线段树和区域树中的联合结构。例如，在求解针对一组与坐标轴平行的矩形的穿刺查询问题时，或是针对形如$[x : x'] \times [y : y'] \times [z : +\infty)$的待查询区域进行区域查找时，这都将会很有用处。

10.5 习题

习题 10.1 第 10.1 节解决了以下问题：在如图 10-26 所示的一组水平线段中，找出与某一给定垂直线段相交的所有线段。为此，我们使用了区间树，并采用优先查找树来实现其联合结构。实际上，还有其它的方法。

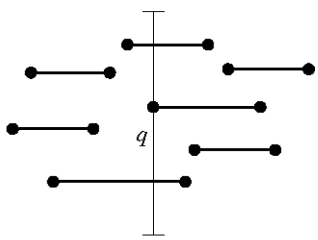


图10-26 一组水平线段

比如，可以按照 y-坐标的次序，将所有线段组织成一棵一维区域树——这样，就可以找出其 y-坐标落在待查询线段（query segment）的 y-区域之内的所有线段。这样找出来的线段，既不可能处于待查询线段上方，也不会处于下方。尽管如此，它们仍然可能会完全处于待查询线段的左侧或者右侧。这些线段，可以表示为 $O(\log n)$个正则子集的并。对于其中每个正则子集，都可以（将其中的线段）按照 x-坐标组织成一棵区间树，并将其作为（对应节点的）联合结构——这样，就可以从中找出真正与待查询线段相交的那些线段。

a. 试用伪代码描述相应的算法。

b. 试证明：该数据结构的确能够正确回答每次查询。

c. 该结构预处理时间、占用存储量以及查询时间的上界各是多少？试给出证明。

习题 10.2 设 P 为由平面上 n 个点组成的一个集合，其中各点已按 y-坐标排序。试证明：只要 P 已经如此排序，就可以在 $O(n)$时间内将 P 中各点组织为一棵优先查找树。

习题 10.3 优先查找树算法的描述中，假定所有点的坐标互异。我们也曾指出，只要采用第 5.5 节所介绍的合成数，就可以消除这种限制。试证明：利用合成数技术，的确可以（在

不做任何限制的情况下）实现优先查找树的构造和查询过程中所需的所有基本操作。

习题 10.4 如果是针对一组互不相交的线段进行截窗查询，就可以将任务一分为二：首先，针对所有的端点进行一次区域查找；随后，分别针对截窗的四条边，各进行一次相交查询。试说明，如何才能不致使同一线段被报告多次。为说明清楚，你需要列举出一条方向任意的线段可能与查询窗口相交的所有情况。

习题 10.5 本题的要求是：说明如何在$O(n\log n)$时间内，构造出〖定理 10.13〗中的数据结构。在那里，我们通过将各线段按照垂直方向的次序组织为一棵二分查找树，以实现联合结构。如果事先已经知道（各线段的）垂直次序，就可以在线性时间内构造出每个联合结构。因此，剩下的问题就是：如何在总共不超过$O(n\log n)$时间内，对各正则子集中的线段进行排序。

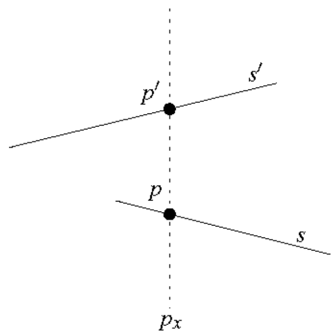


图10-27 不相交线段之间的上下次序

设S为由平面上一组共n条互不相交的线段。对于其中任意两条线段s和$s' \in S$，我们称"s居于s'下方"（记作$s \prec s'$）的充要条件是，分别存在两个点$p \in s$和$p' \in s'$，使得$p_x = p'_x$且$p_y < p'_y$（如图 10-27 所示）。

a. 试证明：在S中定义的$\prec$关系，是一个无环关系（acyclic relation）。也就是说，你需要证明：可以将S中的线段排成一个序列$s_1, s_2, \ldots, s_n$，使得对任何$i > j$都有$s_i \nprec s_j$。

b. 试给出一个算法，在$O(n\log n)$时间内构造出这样一个序列。提示：通过平面扫描，找出沿垂直方向相邻的所有线段；根据这些线段之间的邻接关系（及其垂直高度），将它们组织成一幅有向图；最后，对这幅图实施拓扑排序。

c. 借助这一无环序，如何才能得出线段树中各正则子集所对应的有序列表？试对此做出说明。

习题 10.6 设 I 为实轴上的一组区间。我们希望对给定的任一待查询点，都能够在$O(\log n)$时间内统计出 I 中包含该待查询点的区间数目。这样，查询时间将与实际包含该待查询点的线段数目无关。

a. 试基于线段树，给出解决这个问题的一种数据结构，要求其占用的存储量不得超过$O(n)$。针对该数据结构对应的预处理时间和查询时间，试分别进行分析。

b. 试基于区间树，给出解决这个问题的另一种数据结构。要求将与区间树中各节点相关联的列表，替换为其它的结构。针对该数据结构所占用的存储空间、对应的预处理时间和查询时间，试分别进行分析。

c. 试基于简单的二分查找树，再给出解决这个问题的一种数据结构，要求其占用的存储量不得超过 $O(n)$，查询时间不得超过 $O(\log n)$。（这就是说，实际上根本不需要借助线段树结构，也能有效地解决这一问题。）

习题 10.7 a. 我们的目标是解决如下的查询问题：如 图 10-28 所示，在平面上给定一组共n条互不相交的线段S，对于任何一条发自点(q_x, q_y)、垂直向上直到无穷远的射线，从S中找出与该射线相交的所有线段。

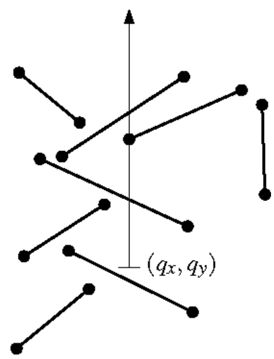


图10-28 与某条射线相交的所有线段

试给出解决这个问题的一种数据结构，要求其占用的存储量不得超过 $O(n\log n)$，而且查询时间不得超过 $O(\log n + k)$，其中 k 为实际被报告出来的线段数目。

b. 接下来，假设我们只希望找出查询射线所穿过的第一条线段。试给出解决这个问题的一种数据结构，要求其占用的期望存储量不超过 $O(n)$，期望查询时间不超过 $O(\log n)$。提示：采用轨迹法。

习题 10.8 可以利用线段树来实现多层次数据结构。

a. 设 R 为平面上一组共 n 条与坐标轴平行的矩形。试给出存储 R 的一种数据结构，要求对于任一待查询点 q，都能够有效地从 R 中报告出包含 q 的所有矩形。针对该数据结构所占用的存储量以及查询时间，试分别进行分析。提示：将所有矩形对应的 x-区间组织成一棵线段树；再利用适当的某种联合结构，来组织线段树中所有节点各自对应的正则子集。

b. 将上述数据结构推广至 d-维空间。此时，给定的是一组与坐标轴平行的超矩形（hyperrectangle）——亦即，形如$[x_1 : x_1'] \times [x_2 : x_2'] \times \ldots \times [x_d : x_d']$的多胞体（polytope）——要求从中找出包含给定待查询点的所有超矩形。针对该数据结构所占用的存储量以及查询时间，试分别进行分析。

习题 10.9 设 I 为实轴上的一组区间。我们希望将这些区间组织成某种数据结构，使得对于任一

区间[x : x']，都可以高效地从 I 中找出包含于该区间之内的所有区间。试给出一个解决这一问题的数据结构，其占用的存储量不得超过 $O(n\log n)$，而且查询时间不得超过 $O(\log n + k)$，其中 k 为实际被报告出来的区间数目。提示：利用区域树。

习题 10.10 与上题一样，这里的输入也是实轴上的一组区间 I，但是问题稍有不同：对于任一给定的区间[x : x']，我们希望能够有效地从 I 中找出包含该区间的所有区间。试给出一个解决这个问题的数据结构，要求其占用的存储量不得超过 $O(n)$，而且查询时间不得超过 $O(\log n + k)$，其中 k 实际被报告出来的区间数目。提示：利用优先查找树。

习题 10.11 二维区域查找问题还有其它的解法，比如，考虑下面的方法：按照 x-坐标，将所有点组织成一棵平衡二分查找树。对树中的任一节点 v，考虑以 v 为根节点的子树，将存储于该子树中的所有点构成一个集合，记之为 P(v)。对于每个节点 v，我们将 P(v)分别组织成两棵优先查找树 $\mathcal{T}_{left}(v)$和 $\mathcal{T}_{right}(v)$——$\mathcal{T}_{left}(v)$支持左端无界的区域查找;$\mathcal{T}_{right}(v)$支持右端无界的区域查找。

针对区域$[x : x'] \times [y : y']$的查询过程如下。首先考虑分别通往 x 和 x'的两条搜索路径，找到它们的分叉位置，设该节点为 v_{split}。接下来，在树 $\mathcal{T}_{right}(lc(v_{split}))$中对区域$[x : +\infty) \times [y : y']$进行查询，并在树 $\mathcal{T}_{left}(rc(v_{split}))$中对区域$(-\infty : x'] \times [y : y']$进行查询。这样，就找出了所有的答案（并不需要继续在树中进行搜索！）。

a. 试证明：利用这种数据结构，的确可以正确地解决区域查找问题。

b. 该数据结构的预处理时间、存储量以及查询时间各是多少？试给出证明。

习题 10.12 a. 我们在第 10.3 节中曾给出过一个算法，将一段区间插入到线段树中（假定区间的两个端点已经在主树中出现了）。试说明，同样可以在 $O(\log n)$时间内删除一个区间。（为此，需要维护一些附加的信息。）

b. 在平面上，设$P = \{p_1, ..., p_n\}$为一组共n个点，$R = \{r_1, ..., r_n\}$为一组共n个可能相交的矩形。试给出一个算法，从P中找出满足$p_i \in r_j$的所有配对(p_i, r_j)。要求算法的运行时间不超过$O(n\log n + k)$，其中k为实际被报告出来的配对数目。

11

凸包： 混合物

从油井中抽出的原油，是包含多种成份的混合物；而采自不同油田的原油，各种成份的比例也不尽相同。为使原油中的各种成份达到某种特定的比例，以便于进一步的加工提炼，通常采用的一种做法，就是将来自不同油井的原油适当地混合起来。

来看一个实例。为简化起见，我们只关心产品中的两类成份——不妨称之为 A 和 B。假设有内含这两类成份的两种混合物 ξ_1 和 ξ_2——ξ_1 中含有 10%的 A 成份，35%的 B 成份；ξ_2 中含有 16%的 A 成份，20%的 B 成份。我们还进一步地假设：我们真正所需要的混合物，应该含有 12%的 A 成份和

30%的 B 成份。那么，利用已有的这两种混合物，能否兑制出我们所需的混合物呢？可以，只要将它们按照 2 : 1 的比例混合起来，就可以达到目的。然而，若我们所需要的是另一种含有 13%A 成份、22%B 成份的混合物，则无论按照何种比例来混合 ξ_1 和 ξ_2，都将是徒劳的。不过，要是同时还有第三种混合物 ξ_3，该混合物含有 7%的 A 成份、15%的 B 成份，那么只要将 ξ_1、ξ_2 和 ξ_3 按照 1 : 3 : 1 的比例混合起来，也可以达到目的。

这类问题，与几何有甚关系？其实，如 图 11-1 所示，只要将ξ_1、ξ_2 和ξ_3 这三种已有的混合物分别表示为平面上的点（具体地，令$p_1 := (0.1, 0.35)$、$p_2 := (0.16, 0.2)$和$p_3 := (0.07, 0.15)$），其间的关系就一目了然了。

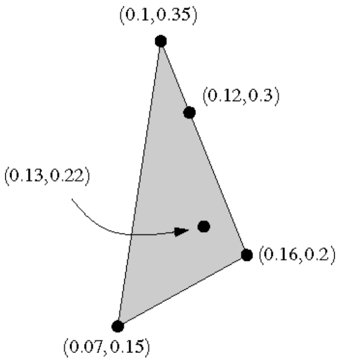


图11-1 三种混合物能够“勾兑”出的混合物，对应于一个三角形内的所有点

将 ξ_1 和 ξ_2 按照 2 : 1 的比例混合，得到的混合物就对应于点 $q := (2/3) \cdot p_1 + (1/3) \cdot p_2$。该点落在线段$\overline{p_1p_2}$上；另外，若用 dist(. , .)表示两点之间的距离，则该点还满足 $\mathrm{dist}(p_2, q) : \mathrm{dist}(q, p_1) = 2 : 1$。一般而言，按照不同比例混合 ξ_1 和 ξ_2，可以“勾兑”出与线段$\overline{p_1p_2}$上任何一点相对应的混合物。如果共三种混合物，就可以“勾兑”出与三角形 $p_1p_2p_3$ 内任何一点相对应的混合物。例如，将 ξ_1、ξ_2 和 ξ_3 按照 1 : 3 : 1 的比例混合起来所得到的混合物，就与点$(1/5) \cdot p_1 + (3/5) \cdot p_2 + (1/5)p_3 = (0.13, 0.22)$相对应。

如果共有 n 种混合物可用，$n > 3$，它们分别对应于点 $p_1, \ldots, p_n$，那么情况又将如何？假设我们按照 $l_1 : l_2 : \ldots : l_n$ 的比例将它们混合起来。令 $L := \sum_{j=1}^{n} l_j$，$\lambda_i := l_i/L$。我们注意到：

$$\text{对任何 } i\text{，都有}\lambda_i \geq 0\text{；}\sum_{i=1}^{n}\lambda_i = 1$$

只要按照如此确定的比例，将已有的这些混合物混合起来，所得到的混合物就将对应于点

$$\sum_{i=1}^{n}\lambda_i p_i$$

如果这里的λ_i的确满足上面的条件——每个λ_i都非负，而且其总和为 1——那么这种线性组合（linear combination）就被称为**凸组合**（convex combination）。第 1 章曾经将一个点集的凸包定义为

“包含这一点集的最小凸集”——或者更准确地，为“包含这一点集的所有凸集的公共交集”。可以证明：任一点集的凸包，都恰好是由这些点可能的所有凸组合构成的集合。因此，为了判断某种所需的混合物能否由已有的几种混合物“勾兑”出来，只要分别找出这些混合物所对应的点，再构造出它们的凸包，最后判断所需混合物所对应的点是否落在该凸包内部。

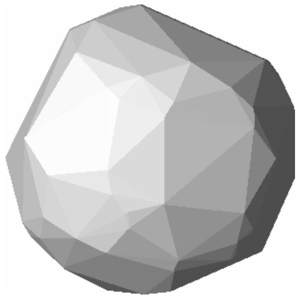


图11-2 混合物包含d种成份时，可“勾兑”出混合物对应于一个d维凸多胞体

那么，要是我们所关心的混合物的成份不止两类，情况又将如何呢？上面的结论依然成立；不过，此时需要进入更高维的空间。更准确地说，如 图 11-2 所示，倘若需要考虑d种成份，就必须将每一种混合物表示为d-维空间中的一个点。任意给定若干种混合物，由它们各自对应的点所构成的凸包，是一个凸多胞体——其内部的点，对应于所有可以“勾兑”出来的混合物。

凸包（尤其是三维空间中的凸包），在众多应用中都能发挥作用。例如在计算机动画领域，这种几何结构就可以被用来加速碰撞检测。任意给定两个物体 P_1 和 P_2，我们希望检测它们是否相交。如果在大多数情况下问题的答案都是否定的，那么使用下面的方法就会很合算。将这两个物体分别近似为$\hat{P}_1$和$\hat{P}_2$，这两个新的物体不仅分别包含了原先的两个物体，而且相对来说更为简单。现在，如果要判断 P_1 和 P_2 是否相交，可以首先判断$\hat{P}_1$和$\hat{P}_2$是否相交；只有在这对新的物体相交时，才需要进一步去检测原先的那对物体是否相交（假设后一检测更加耗时）。

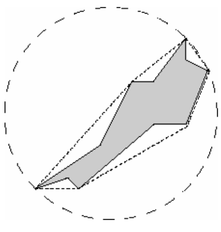


图11-3 使用包围球近似几何体的效果往往不好

那么，应该将原先的那对物体近似到何种程度呢？这里需要做个折衷。一方面，我们希望它们尽可能地简单——这样，求交检测的代价才会更低。而另一方面，新的物体越是简单，其对原先物体的近似程度也就越低，相应地，我们需要对原来的物体进行检测的可能性也将更大。在所有的可

能中，一个极端的选择就是采用包围球（bounding sphere）——虽然球体之间的求交计算非常简单，但是对于许多的物体来说，使用球体进行近似的效果并不算好。除包围球之外，另一种选择是使用凸包——尽管与球体相比，凸包之间的求交计算更为复杂，但是相对于非凸的物体而言，还是要简单得多；更重要的是，大多数的物体都可以通过凸包来更好地近似。

11.1 三维凸包的复杂度

我们在第 1 章中已经看到，对于由平面上任意n个点组成的集合P，其凸包都是一个凸多边形（convex polygon）[①]，而且该多边形的顶点全部来自于P。因此，该凸包至多拥有n个顶点。在三维空间中，类似的结论依然成立：由任意n个点组成的集合P，其凸包都是一个凸多胞体，而且该多胞体的顶点全部来自于P。因此，该凸包将由不超过n个顶点确定。在平面情况中，根据顶点数目的上界（upper bound），马上就可以得出一个结论：凸包本身的复杂度是线性的——因为，在任何一个平面多边形中，边的数目与顶点的数目必然相等。然而在三维空间中，这一点将不再成立。实际上，一个多胞体中所含的边可能会多于顶点。不过还算幸运，正如下面这则定理将要指出的，在任意凸多胞体中，边与小平面的数目仍不会相差太多。（按照正规的定义，如图 11-4 所示，所谓凸多胞体的一张**小平面**，就是由其边界上的共面点所组成的极大集合。凸多胞体的每一张小平面，都必然是一个凸多边形。而凸多胞体的每一条边，都是它的某张小平面的边。）

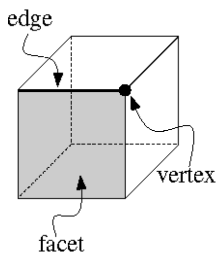


图11-4 凸多胞体的组成

〖定理 11.1〗

设P为包含 n 个顶点的任一凸多胞体。则P中所含的边不会超过 3n - 6 条，所含的小平面不会超过 2n - 4 张。

〖证明〗

首先，让我们回忆起欧拉公式。该公式指出：对于任何连通的平面图，若其中包含 n 个节点、n_e 条边和 n_f 张面，则如下关系必然成立：

① 如果不考虑退化情况的话。——译者

$$n - n_e + n_f = 2$$

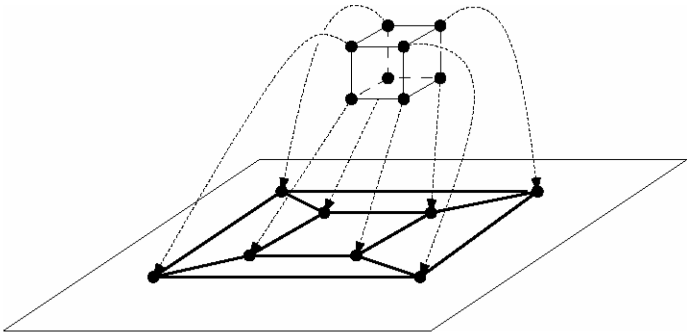


图11-5 将一个立方体看作一个平面图。
请注意，其中的某张小平面将被映射为图中的一张无界面

对于任何凸多胞体，都可以按 图 11-5 所示的方法，将其边界看作一幅平面图。于是，凸多胞体的顶点数、边数以及面数也同样满足上面的关系。（实际上，最初的欧拉公式所针对的本来就是多胞体，而不是平面图。）在与*P*对应的图中，每张面都由至少三条弧围成；而且，每一条弧都与两张面相关——如此就有 $2n_e \geq 3n_f$。将这一不等式代入欧拉公式，就得到：

$$n + n_f - 2 \geq 3n_f / 2$$

于是，$n_f \leq 2n - 4$。只要再应用一次欧拉公式，就可以得到 $n_e \leq 3n - 6$。当然，有可能所有的小平面都是三角形，即所谓的单纯多胞体（simplicial polytope）——在这种特殊情况下，因为 $2n_e = 3n_f$，所以上述关于边及小平面数目的上界都将成为确界。 ❑

对于所谓的亏格（genus）为零（即不含孔洞和通道）的非凸多胞形，〖定理 11.1〗依然成立。对于亏格更大的多胞体，也存在类似的上界。不过，鉴于本章讨论的是凸多边形，我们就不对（非凸）多胞体做严格的定义了——若要针对非凸的情况证明本定理，的确需要先给出这一定义。

我们在此前已经观察到，三维点集P的凸包是一个凸多胞体，而且其顶点全部来自于集合P。只要将这一观察结论与〖定理 11.1〗结合起来，就可以得出如下结论：

〖**推论** 11.2〗

在三维空间中，任意 n **个点的凸包的复杂度为** *O*(n)。

11.2 构造三维凸包

设P为由三维空间中任意n个点组成的一个集合。与此前在第 4、6 和 9 章中的做法一样，这里也将借助随机增量式算法，来构造P的凸包*CH*(P)。

我们的递增式构造过程的第一步是，在P中选出不共面的四个点——这样，它们的凸包必是一个四面体（tetrahedron）。具体做法如下。任取P中的两点p_1和p_2。然后，依次将集合P中的各点，与由p_1和p_2确定的那条直线进行比较，直到出现一个不落在该直线上的点p_3。接下来，继续将P中的各点，与由p_1、p_2和p_3确定的平面进行比较，直到出现一个不落在该平面上的点p_4。（要是找不到这样的四个点，就说明P中的所有点都共面。果真如此，就可以借助第 1 章所介绍的算法，来计算该集合的凸包。）

接下来，还要将其余各点的次序随机打乱，将它们排成 $p_5, \dots, p_n$。我们将按照这一随机的次序，逐一处理各点；在此过程中，还将动态地维护凸包。对于任何一个整数 $r \geq 1$，令 $P_r := \{p_1, \dots, p_r\}$。该算法反复循环，每循环一次，都要将点 p_r 加入到 P_{r-1} 的凸包中去——也就是将 $\mathcal{CH}(P_{r-1})$转化为 $\mathcal{CH}(P_r)$。其间可能出现两种情况：

- 若 p_r 落在 $\mathcal{CH}(P_{r-1})$的内部或者边界上，则 $\mathcal{CH}(P_r) = \mathcal{CH}(P_{r-1})$，无需做任何计算。

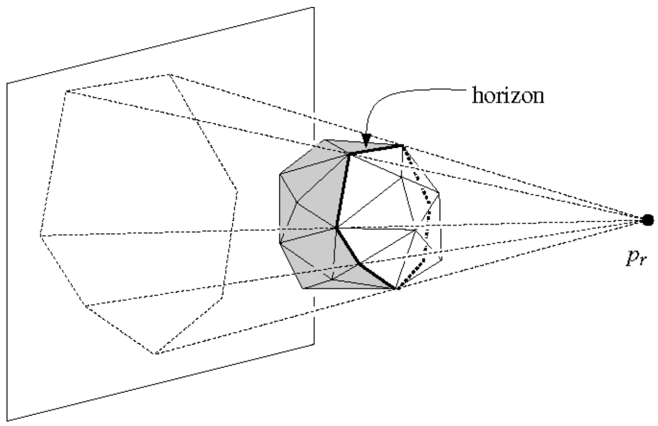


图11-6 多胞体的地平线

- 现假设p_r落在$\mathcal{CH}(P_{r-1})$之外。请试想着站在点p_r的位置，朝$\mathcal{CH}(P_{r-1})$看去。你应该能够看到$\mathcal{CH}(P_{r-1})$的某些小平面——这些小平面都在正面；但是另外的那些就无法看到了，因为它们都在背面。在$\mathcal{CH}(P_{r-1})$的表面上，所有那些可见的小平面组成了一个连通的区域，称作p_r在$\mathcal{CH}(P_{r-1})$上的**可见区域**（visible region）。这个区域，是由$\mathcal{CH}(P_{r-1})$的某些边组成的一条封闭折线围成的，我们称这条折线为p_r在$\mathcal{CH}(P_{r-1})$上的地平线（horizon）。正如可以从 图 11-6 中看到的那样，如果以p_r为中心将地平线投影到某平面上，得到的影像恰好就是$\mathcal{CH}(P_{r-1})$到同一平面的投影的边界。从几何意义上讲，“可见”究竟意味着什么呢？

  如 图 11-7 所示，考察$\mathcal{CH}(P_{r-1})$上任一小平面f所在的那张平面h_f。这张平面定义了两个闭的半空间——根据$\mathcal{CH}(P_{r-1})$的凸性，它必然完全包含于其中的某个闭半空间之中；如果一个点来自于h_f另一侧的那个开空间，它就会与面f可见。

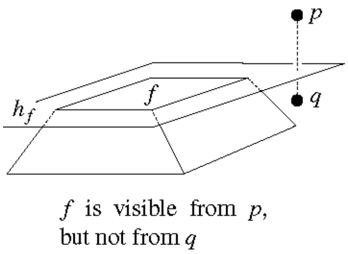


图11-7 平面h_f为$CH(P_{r-1})$贡献一张小平面f（f与p可见，但与q不可见）

在将 $CH(P_{r-1})$转化到 $CH(P_r)$的过程中，p_r的地平线扮演了一个重要的角色。为完成这一转化，原先 $CH(P_{r-1})$表面的哪些部分需要被保留下来，哪些又需要被替换掉呢？实际上，需要保留的部分，就是与 p_r 不可见的那些小平面；需要被替换掉的部分，就是与 p_r 可见的那些小平面。而 p_r 的地平线，恰好对应于这两部分表面之间的边界。我们必须在 p_r 与其对应的地平线之间联结出若干新的小平面，以替换所有可见的小平面。

那么，应该如何来表示（三维）空间中的凸包呢？在做进一步的介绍之前，必须首先回答这一问题。正如我们已经在此前所注意到的，三维凸多胞体的边界可以被看作是一幅平面图。既然如此，如 图 11-8 所示，在这里就可以利用双向链接边表（doubly-connected edge list）来存储凸包（第 2 章中曾建立起这种数据结构，来存储平面的子区域划分（subdividsion））。

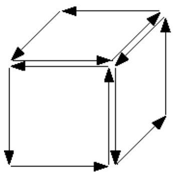


图11-8 将三维凸包表示为双向链接边表

唯一的不同之处在于，这里的顶点都是三维（空间中）的点。这里仍将遵守原先的习惯——若从多胞体的外部看去，沿着每一张面的边界，所有的半边都构成一个逆时针方向的环路。

现在回过头来，继续讨论如何将p_r引入到凸包当中。也就是说，已知一个对应于$CH(P_{r-1})$的双向链接边表，如何将其转换为一个与$CH(P_r)$相对应的双向链接边表。为此，如 图 11-9 所示，需要将$CH(P_{r-1})$中与p_r可见的所有小平面所对应的信息，从原先的双向链接边表中删除掉；然后，需要生成一些新的小平面，将p_r与其地平线联接起来，并且将这些新的小平面所对应的信息存储到双向链接边表中。只要我们已经在$CH(P_{r-1})$中找出了所有与p_r可见的小平面，上述计算并不难实现。实际上，这部分计算的复杂度，将线性正比于需要删除的小平面数。在引入一批新的小平面之后，有一点细微之处需要格外留意：必须检查一下，所生成的小平面中是否有共面的情况。如 图 11-10 所示，当p_r与$CH(P_{r-1})$的某张小平面f共面时，就会出现这种情况。按照此前对可见性的定义，这样的面f与p_r并不可见。于是，f会得以保留下来；但是，由于f边界上的某些边属于p_r的地平线，故算法会生成若干三角形，并

将p_r与这些边联接起来。这些三角形必然与f共面，因此必须将这些面与原先的f合并起来，构成统一的一张小平面。

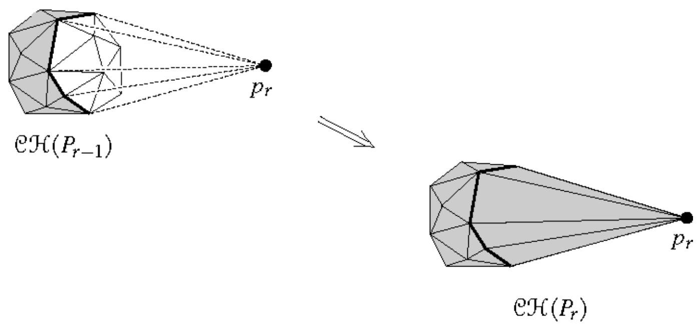


图11-9 将新的一个点引入到凸包中

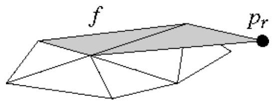


图11-10 引入的小平面可能共面

讨论到现在，我们一直忽略了一个问题：如何才能在$\mathcal{CH}(P_{r-1})$的边界上，找出与 p_r 可见的所有小平面呢？当然，只要逐一检查各张小平面，就可以完成这一任务。在检测每张小平面时，只要看看 p_r 究竟落在该小平面所在平面的哪一侧，就可以做出判断——因此，每一次测试只需要常数的时间。也就是说，可以在$O(r)$时间内，确定所有的可见小平面。按照这种方法，算法总体的复杂度将是$O(n^2)$。能否更快呢？下面就来介绍一种更好的方法。

这一方法的诀窍在于，我们可以提前进行计算。除了当前点集的凸包之外，我们还要维护一些附加的信息——借助于这些信息，很容易就可以找出那些可见的小平面。具体来说，对于当前凸包$\mathcal{CH}(P_r)$的每一张小平面 f，都要维护一个集合 $P_{conflict}(f) \subseteq \{p_{r+1}, p_{r+2}, \ldots, p_n\}$，这个子集是由与 f 可见的那些点组成的。反过来，对于每一个点 p_t（$t > r$），也要维护一个集合 $F_{conflict}(p_t)$，这个集合是由 $\mathcal{CH}(P_r)$中所有与 p_t 可见的小平面组成。我们断言：点 $p \in P_{conflict}(f)$与小平面 f 发生**冲突**。这是因为，p 和 f 不可能在凸包中“和平”相处——一旦有一个点 $p \in P_{conflict}(f)$加入到凸包中，小平面 f 就必须被删除。我们将 $P_{conflict}(f)$和 $F_{conflict}(p_t)$称作冲突列表（conflict list）。

我们将借助所谓的冲突图（conflict graph）来记录冲突的情况，这幅图记作$\mathcal{G}$。如 图 11-11 所示，所谓的冲突图，是一幅二部图（bipartite graph）。也就是说，其中的节点被划分为两个子集——在其中的一个子集内，各节点分别对应于P中尚未被插入的每一个点；而在另一个子集中，各节点分别对应于当前凸包的各张小平面。发生冲突的任何点和小平面之间，都通过一条弧相互联接。具体而言，只要$\mathcal{CH}(P_r)$中的某张小平面f与$p_t \in P$可见（$r < t$），就会有一条弧将它们（分别对应的节点）联接起来。借助冲突图$\mathcal{G}$，我们就可以对任意给定的一个点p_t，报告出集合$F_{conflict}(p_t)$，而所需的时间将线性正比于该集合的规模。类似地，对任意给定的一张小平面f，也可以在线性的时间内报告出集合

$P_{conflict}(f)$。这就是说，为了将p_r插入到$\mathcal{CH}(P_{r-1})$中，只需要通过在$\mathcal{G}$中查找$F_{conflict}(p_r)$，即可找出（与p_r）可见的所有小平面，然后将它们删除，并代之以由p_r与地平线相联接而生成的小平面，从而得到更新后的凸包。

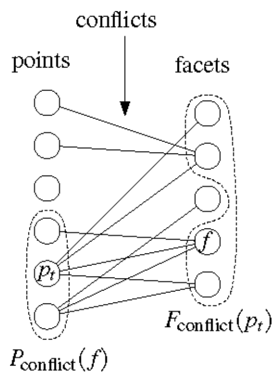


图11-11 冲突图

可以在线性时间内，对 $\mathcal{CH}(P_4)$的冲突图 $\mathcal{G}$ 进行初始化：只要依次检查 P 中所有的（n-4 个）点，就可以判断出，它们分别与 $\mathcal{CH}(P_4)$的 4 张面中的哪些可见。

在引入点 p_r 之后，为了更新 $\mathcal{G}$，首先找出凸包 $\mathcal{CH}(P_{r-1})$上所有随之消失的小平面，删去与之对应的节点以及与之关联的弧。它们都是与 p_r 可见的小平面——在 $\mathcal{G}$ 中，它们恰好就是 p_r 的所有邻居，所以这很容易做到。我们还要删去与 p_r 对应的节点。然后，对应于新生成的每一张小平面（它们将 p_r 与地平线联接起来），都要在 $\mathcal{G}$ 中新增一个节点。关键的一步，是要分别计算出这些新的小平面各自对应的冲突列表。不需要对其它的冲突进行更新——只要 p_r 的引入没有影响到小平面 f，f 所对应的冲突集（conflict set）也就不会有任何变化。

由于 p_r 的插入而新生成的每张小平面，一般都是三角形，除非它由于与已有的某张小平面共面而需要进行合并。后一类小平面的冲突列表可直接得到——经合并后，原有小平面所属的平面不会发生改变，因此小平面的冲突列表应该与原有小平面相同。因此，我们在 $\mathcal{CH}(P_r)$上任取一张与 p_r 关联的新三角面 f，并对其进行考察。假设点 p_t 与 f 可见。于是，p_t 肯定也能看到 f 上与 p_r 相对的边 e。

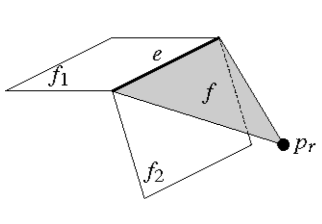


图11-12 p_r地平线上的边e

如 图 11-12 所示，这条边e，就是p_r的地平线上的一条边，因此它在$\mathcal{CH}(P_{r-1})$中必然已经出现了。因为$\mathcal{CH}(P_{r-1}) \subset \mathcal{CH}(P_r)$，所以在$\mathcal{CH}(P_{r-1})$中，$p_t$也必然与e可见。若是这样，在$\mathcal{CH}(P_{r-1})$中与e关联的那两张小平面（记作$f_1$ 和f_2），其中之一必然与p_t可见。这就意味着，只要取出f_1 和f_2 的冲突列表，逐一检

查其中各点，就能构造出f的冲突列表。

此前曾经指出，我们使用了双向链接边表结构来存储凸包。因此，所谓的“改变凸包”，就是“改变双向链接边表结构中的信息”。然而，在用下列伪代码来描述凸包算法时，为了使代码保持简明，其中对双向链接边表的所有显式引用都被略去了。

```
算法 CONVEXHULL(P)
输入：三维空间中 n 个点组成的集合 P
输出：P 的凸包 CH(P)
1.  在 P 中找出四个点 p1、p2、p3 和 p4，构成一个四面体
2.  C ← CH({p1, p2, p3, p4})
3.  任取其余各点的一个随机排列：p5, p6, …, pn
4.  对图 G 初始化，令 G = {(pt, f) | f 为 C 上的小平面，t > 4}
5.  for r ← 5 to n
6.    do(* 将 pr 插入到 C 中 *)
7.      if (Fconflict(pr)非空)   (* 即，pr 落在 C 外部 *)
8.        then 将 Fconflict(pr)中所有的小平面从 C 中删除
9.             沿着（恰好由 Fconflict(pr)中各小平面组成的）pr 可见区域的边界行进
             将沿地平线的各边组织成一个有序表 L
10.            for (所有 e∈L)
11.              do 通过生成一张三角形小平面 f，将 e 与 pr 联接起来
12.                if (和 f 相邻于 e 的小平面 f'与 f 共面)
13.                  then 将 f 与 f'合二为一，
                        合并后小平面的冲突列表与 f'的相同
14.                  else (* 确定 f 引起的冲突 *)
15.                       在 G 中生成一个对应于 f 的节点
16.                       考虑原先凸包上与 e 关联的那两张小平面，
                            令为 f1 和 f2
17.                       P(e) ← Pconflict(f1) ∪ Pconflict(f2)
18.                       for (所有点 p∈P(e))
19.                         do 若 f 与 p 可见，则将(p, f)加入到 G 中
20.            在 G 中删除以下点和弧：
                 a. 对应于 pr 的节点、
                 b. 与 Fconflict(pr)中各小平面对应的节点、以及
                 c. 与之关联的所有弧
```

21. **return**($\mathcal{C}$)

11.3 *分析

与通常对随机增量式算法的分析过程一样，我们首先要对结构变化的期望量做出界定。就凸包算法而言，也就是要对整个算法过程中所生成小平面的总数做出估计。

〖引理 11.3〗

由算法 CONVEXHULL 所生成的小平面，总数的期望值不超过 6n-20。

〖证明〗

该算法起始于一个四面体，它由四张小平面构成。在算法的第 r 轮迭代中，如果 p_r 落在 $\mathcal{CH}(P_{r-1})$的外部，就要通过生成若干张三角形小平面，将 p_r 与 $\mathcal{CH}(P_{r-1})$上的地平线联接起来。那么，这些新生成的小平面的期望数目是多少呢？与对此前各随机算法（randomized algorithm）的分析一样，我们依然要借助后向分析。考察 $\mathcal{CH}(P_r)$，然后假想着删除顶点 p_r；随着 p_r 的删除而消失的小平面的数目，恰好等于由于在 $\mathcal{CH}(P_{r-1})$中加入 p_r 而生成的小平面的数目。而消失的小平面，恰好就是在 $\mathcal{CH}(P_r)$中与 p_r 相关联的那些小平面，它们数目等于与 p_r 相关联的边数。这一数目，被称作“p_r 在 $\mathcal{CH}(P_r)$中的度数”，记作 $\deg(p_r, \mathcal{CH}(P_r))$。这样，我们的任务就是界定 $\deg(p_r, \mathcal{CH}(P_r))$的期望值。

根据〖定理 11.1〗，由r个顶点构成的凸多胞体至多含有 3r-6 条边。既然$\mathcal{CH}(P_r)$是一个由不超过r个顶点构成的凸多胞体，这就意味着，其中所有顶点的度数之和不会超过 6r-12。于是，平均度数将不超过 6-12/r。因为我们是按照随机次序进行处理的，似乎p_r的期望度数应该不会超过 6-12/r。然而，我们还是要更加仔细——在确定一个随机次序时，其中有四个点已经是固定了的，因此更确切地说，p_r是$\{p_5, \ldots, p_r\}$（而不是Pr）中的一个随机元素。考虑到$p_1, \ldots, p_4$的度数和不小于 12，$\deg(p_r, \mathcal{CH}(P_r))$的期望值应该按如下界定：

$$E[\deg(p_r, \mathcal{CH}(P_r))]$$

$$= \frac{1}{r-4} \cdot \sum_{i=5}^{r} \deg(p_r, \mathcal{CH}(P_r))$$

$$\leq \frac{1}{r-4} \cdot (\{\sum_{i=1}^{r} \deg(p_r, \mathcal{CH}(P_r))\} - 12)$$

$$\leq \frac{6r-12-12}{r-4}$$

$= \quad 6$

算法 ConvexHull 所生成的小平面的期望数目，就是起始时的小平面数目（即 4），再加上在依次加入 $p_5, \ldots, p_n$ 的过程中所生成小平面的期望数目。因此，所生成小平面的期望数目就等于：

$$4 + \sum_{i=5}^{n} E[\deg(p_r, \mathcal{CH}(P_r))] \quad \leq \quad 4 + 6(n-4) \quad = \quad 6n - 20$$

❑

在对结构的变化总量做出界定之后，就可以进一步来界定算法运行时间的期望值。

〖引理 11.4〗

对于由 $\mathbb{R}^3$ 中任意 n 个点组成的集合 P，算法 ConvexHull 都可在 $O(n\log n)$的期望时间内构造出 P 的凸包。这里的期望值，是相对于算法所采用的随机排列而言的。

〖证明〗

在主循环之前的那些步骤，肯定可以在 $O(n\log n)$时间内完成。在算法的第 r 轮迭代中，若 $F_{conflict}(p_r)$是空集（即当 p_r 落在当前凸包的内部或者边界上时），只需要常数时间。

否则，除了第 17~19 行、第 20 行之外，第r轮迭代的主体部分需要$O(card(F_{conflict}(p_r)))$时间（这里的card()表示集合的基数）。例外的这几行所消耗的时间，将在稍后进行界定；我们首先来界定$card(F_{conflict}(p_r))$。我们注意到，所谓的$card(F_{conflict}(p_r))$，就是随着点p_r的引入而被删除的小平面的数目。显然，每张小平面只有在首先被生成后，才能被删除；而且，最多被删除一次。根据〖引理 11.3〗，算法所生成小平面的期望数目为$O(n)$，因此这就意味着删除操作的（期望）总数同样为$O(n)$，即

$$E[\sum_{i=5}^{n} card(F_{conflict}(p_r))] \quad = \quad O(n)$$

接下来，考虑第 17~19 行和第 20 行。第 20 行所需的时间，线性正比于从 $\mathcal{G}$ 中删除的节点和弧的总数。同样地，一个节点或一条弧至多只能被删除一次，我们可以将每次删除所需的时间，归入为生成该节点或弧所需的时间。这样，剩下的工作就是考察第 17~19 行。在第 r 轮迭代中，对于组成地平线的每一条边（即 $\mathcal{L}$ 中的每一条边），这三行都要执行一遍。对应于边 $e\in\mathcal{L}$ 的这部分时间为 $O(card(P(e)))$。因此，第 r 轮迭代中花费在这三行上的时间总共为 $O(\Sigma_{e\in\mathcal{L}} card(P(e)))$。于是，为了界定期望的总体运行时间，我们需要界定出下式的期望值：

$$\sum_{e} card(P(e))$$

这里的求和范围，覆盖在算法的各轮迭代中，曾经在地平线上出现过的所有边。后面将证明，这个数为 $O(n\log n)$——这就说明，总体的运行时间为 $O(n\log n)$。 ❑

为得出支持上述证明的那个上界，要借助第 9 章中的构形空间框架。在这里，域X就是集合P，而构形Δ则对应于凸包上的各边。不过，由于技术上的原因——为了能够正确处理好退化情况——需要在每条边的两侧分别附上一条半边（half-edge）。更确切地说，我们将引入翼（flap）的概念——所谓的一个翼Δ，就是由不共面的四个点所构成的有序四元组（p, q, s, t）。而定义集D(Δ)正是集合{p, q, s, t}。要对毁灭集K(Δ)做出直观解释，将会更困难些。将由p和q确定的直线记作l。对于给定的任何点x，将穿过x、以l为边界的半平面记作h(l, x)。对于给定的任何两点x和y，将起始于x、穿过y的射线记作ρ(x, y)。

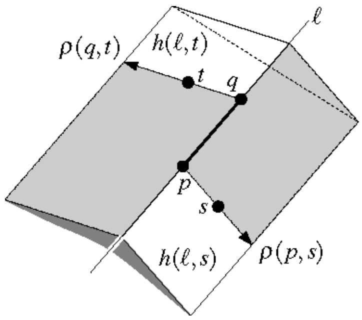


图11-13　点x∈X属于K(Δ)的几种情况

如 图 11-13 所示，任何点x∈X属于K(Δ)，当且仅当x落在如下区域之一的内部：

1. 由半平面 h(l, s)和 h(l, t)确定的三维闭合凸楔形外部；
2. h(l, s)的内部、由射线ρ(p, q)和ρ(p, s)确定的二维闭合楔形的外部；
3. h(l, t)的内部，由射线ρ(q, t)和ρ(q, p)确定的二维闭合楔形的外部；
4. 直线 l 上、线段$\overline{pq}$之外；
5. 射线ρ(p, s)上、线段$\overline{ps}$之外；
6. 射线ρ(q, t)上、线段$\overline{qt}$之外。

对于任一子集S⊆P，都可以定义出一个由活跃构形（active configuration）组成的集合T(S)——这正是我们希望计算的东西——正如第 9 章所规定的：Δ∈T(S)当且仅当D(Δ)⊆S且K(Δ)∩S = ∅。

〖引理 11.5〗

翼$\Delta = (p, q, s, t)$属于$\mathcal{T}(S)$，当且仅当$\overline{pq}$，$\overline{ps}$和$\overline{qt}$都是凸包$\mathcal{CH}(S)$的边，有某张小平面 f_1 与$\overline{pq}$和$\overline{ps}$关联，而且还有另一张小平面 f_2 与$\overline{pq}$和$\overline{qt}$关联。此外，若小平面 f_1 和 f_2 之一与某个点 $x \in P$ 可见，则 $x \in K(\Delta)$。

其证明由读者完成——为此需要精细地考察多点共线或共面的情况，不过，这些都不算困难。

正如你可能预料到的，翼在这里所起的作用，就相当于地平线上的各边。

〖引理 11.6〗

$\Sigma_e card(P(e))$的期望值为 $O(n\mathbf{log}n)$——求和范围，覆盖了出现于该算法各轮迭代中的所有地平线边。

〖证明〗

考虑p_r相对于$\mathcal{CH}(P_{r-1})$的地平线上的任一条边e。令$\Delta = (p, q, s, t)$为满足$\overline{pq} = e$的两个翼中之一。由〖引理 11.5〗，$\Delta \in \mathcal{T}(P_{r-1})$，且在$P \setminus P_r$内的各点中，与关联于e的某张小平面可见的每一个点都属于$K(\Delta)$，因此必有$P(e) \subseteq K(\Delta)$。由此，根据〖定理 9.15〗可知：和式

$$\sum_{\Delta} card(K(\Delta))$$

（这里的求和范围，覆盖了出现于至少一个$\mathcal{T}(P_r)$当中的所有翼Δ）不会超过

$$\sum_{r=1}^{n} 16 \cdot \left(\frac{n-r}{r}\right) \cdot \left(\frac{E[card(\mathcal{T}(P_r))]}{r}\right)$$

该和式中，$\mathcal{T}(P_r)$的基数为$\mathcal{CH}(P_r)$中边数的两倍。因此不会超过6r-12，故可以得到上界：

$$\sum_{e} card(P(e)) \le \sum_{\Delta} card(K(\Delta)) \le \sum_{r=1}^{n} 16 \cdot \left(\frac{n-r}{r}\right) \cdot \left(\frac{6r-12}{r}\right) \le 96 \cdot n \cdot \mathbf{ln} n$$

❑

至此，我们终于完成了对上述凸包算法的分析。分析的结果可以归纳如下：

〖定理 11.7〗

$\mathbb{R}^3$ 中 n 个点的凸包，可在 O(n**log**n)的期望时间内构造出来。

11.4 *凸包与半空间求交

第 8 章已经介绍过对偶变换的概念。对偶变换的威力在于，它使得我们可以从一个新的角度来考虑问题——从这个角度，可以对问题有更为深入的理解。你应该记得，我们将点p的对偶直线记作p^*，将直线l的对偶点记作l^*。对偶变换具有保持关联、保持顺序的性质——$p \in l$当且仅当$l^* \in p^*$；p落在l的上方，当且仅当l^*落在p^*的上方。

我们来仔细看看，凸包在对偶空间中将对应于什么。只考虑平面的情况。设P为任一平面点集。鉴于技术的原因，我们只将注意力集中于其上凸包（upper hull）——记作$\mathcal{UH}$(P)。$\mathcal{UH}$(P)上每一条边的支撑线（supporting line），都从整个P的上方穿过——参见 图 11-14 的左边。所谓的上凸包，即联接于P中最左侧点与最右侧点之间的一条多边形链。（为简化讨论，假定各点x-坐标互异。）

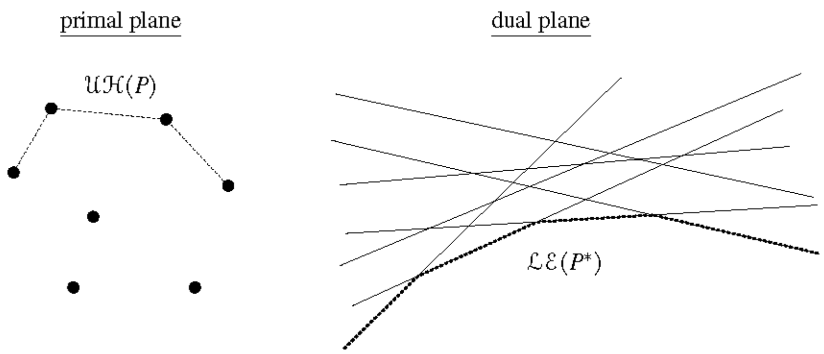


图11-14　上凸包对应于下包络：原平面（左），对偶平面（右）

那么，点$p \in P$何时会作为一个顶点出现在上凸包上呢？出现这种情况，当且仅当存在一条经过p的非垂直线l，使得P中所有的其它点都落在l下方。转换到对偶平面中，这句话可以转述为如下条件：在直线$p^* \in P^*$上存在一个点l^*，使得l^*位于P^*中所有其它直线的下方。如果考察排列$\mathcal{A}(P^*)$，这一条件就意味着：p^*为该排列中唯一的那个底单元（bottom cell）贡献了一条边。这个底单元，就是分别以P^*中各直线为边界、位于直线下方的所有半平面的公共交集。该底单元的边界，是一条x-单调链。如果将P^*中的每条直线都看成是某个线性函数的图像，那么这条链就是所有函数的最小值。正因为此，在任何排列中，底单元的边界往往也被称为是其对应直线集的下包络（lower envelope）。P^*的下包络，记作$\mathcal{LE}(P^*)$——如 图 11-14 的右侧所示。

$\mathcal{UH}$(P)上来自 P 的点，按 x-坐标递增的次序出现。底单元边界上来自 P^*的直线，按斜率递减的次序出现。因为 p^*的斜率等于 p 的 x-坐标，$\mathcal{UH}$(P)上各点自左向右排成的序列，恰好对应于 $\mathcal{LE}(P^*)$中各

边自右向左排列的序列。因此就其本质而言，每个点集的上凸包都等同于某一直线集的下包络。

还有最后一点需要检查。P 中的两个点 p 和 q 构成凸包上的一条边，当且仅当 P 中所有其它的点都落在由 p 和 q 确定的直线下方。在对偶平面上，这个条件将意味着：满足 $r \in P\backslash\{p, q\}$的所有直线 r*，都从直线 p^*和 q*的交点 l^*上方穿过。这正好就是“$p^* \cap q^*$为 $\mathcal{LE}(P^*)$的一个顶点”的条件。

P 的下凸包（lower hull）与 P^*的上包络（upper envelope）之间有何关系呢？（请读者自己给出这些概念的准确定义。）根据对称性，这两个概念同样地互为对偶。

由此可知：为一组下半平面（lower half-plane）——即位于某条非垂直线下方的半平面——计算公共交集的问题，可以转换为计算某个上凸包的问题；而为一组上半平面（upper half-plane）计算公共交集的问题，则可以转化为计算某个下凸包的问题。但是，如果我们需要计算任意的一组半平面 H 的公共交集，又该如何呢？一种显而易见的方法是：先将集合 H 一分为二：H_+为所有的上半平面，H_-为所有的下半平面。然后，通过计算 H_+^*的下凸包，可以得到$\cup H_+$；通过计算 H_-^*的上凸包，可以得到$\cup H_-$。最后，只要对$\cup H_+$和$\cup H_-$进行求交，就可以得到$\cap H$。

然而，有必要这样做吗？既然下包络、上包络对应于上凸包、下凸包，任意一组半平面的公共交集难道不是对应于整个凸包吗？就某种意义而言的确如此。问题在于， 对偶变换无法处理垂线；而且，对于任意两条接近垂直的直线，只要它们的斜率符号相反，那么尽管非常靠近，它们经过映射之后各自对应的点却将相去甚远。凸包的对偶总是由相去甚远的两部分组成，原因正在于此。

的确可以定义出另一种能够支持垂线的对偶变换。不过，为了将该对偶变换应用于给定的一组半平面，需要首先在这些半平面的公共交集中找出一个点。但是我们不能指望总是能够找到这样的一个点。只要我们仍然限于欧氏平面，就不可能找到某一通用的对偶变换，以将任意一组半平面的公共交集转换为一个凸包——因为半平面的公共交集有可能是空集。与这种可能情况相对应的对偶，又将是什么？关于欧氏空间中一组点的凸包，我们的定义总是明确的——它不可能是“空集”。（虽然在有向射影空间（oriented projective space）中，可以完美地解决这个问题，但是这一概念已经超出了本书所涉及的范围。）只有在你能够肯定交集非空，而且已经从中找出了一个点的时候，你才能定义出这样的一个对偶变换，从而将该交集与某个凸包联系起来。

这方面的讨论暂告一段落。重要是，尽管有技术上的复杂性，就其本质而言，凸包与半平面（或三维空间中的半空间）的交集的确是对偶的一对概念。因此，任何一个凸包算法，都可以通过对偶变换，转化为一个构造平面上一组半平面（或者三维空间中一组半空间）的公共交集的算法。

11.5 *再论 Voronoi 图

第 7 章曾介绍过平面点集的Voronoi图（如 图 11-15 所示）。将会令你感到惊讶的是，在三维空

间中，一组上半空间（upper half-space）的公共交集，竟然会与平面Voronoi图有着密切的联系。根据前一节有关对偶变换的结论，这意味着在平面Voronoi图与三维凸包之间也存在着密切的联系。

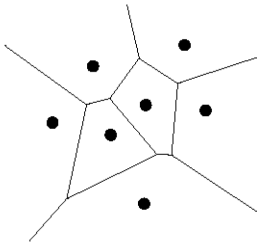


图11-15 Voronoi图

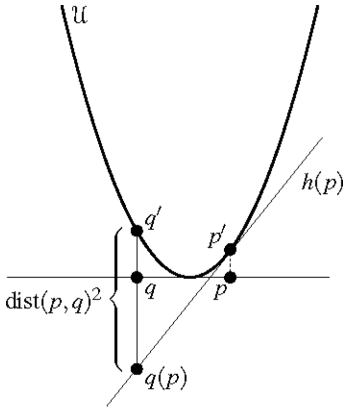


图11-16 借助对偶变换，可以在平面Voronoi图与三维凸包之间建立联系

这与三维空间中单位抛物面的一个神奇性质有关。设$\mathcal{U} := (z = x^2 + y^2)$为单位抛物面，$p := (p_x, p_y, 0)$为平面$z = 0$ 上任意一点。如 图 11-16 所示，考虑穿过p的那条垂线。该直线与$\mathcal{U}$相交于$p' := (p_x, p_y, p_x^2 + p_y^2)$。令h(p)为非垂直的平面$z = 2p_x{\cdot}x + 2p_y{\cdot}y - (p_x^2 + p_y^2)$。请注意到，h(p)必然穿过点p'。现在，考察平面$z = 0$ 上的另一点$q := (q_x, q_y, 0)$。穿过q的那条垂线与$\mathcal{U}$相交于点$q' := (q_x, q_y, q_x^2 + q_y^2)$，而且与h(p)相交于

$$q(p) \quad := \quad (q_x, q_y, 2p_xq_x + 2p_yq_y - (p_x^2 + p_y^2))$$

点 q'到 q(p)的垂直距离等于

$$q_x^2 + q_y^2 - 2p_xq_x - 2p_yq_y + p_x^2 + p_y^2 \quad = \quad (q_x - p_x)^2 + (q_y - p_y)^2 \quad = \quad \text{dist}(p, q)^2$$

因此，平面 h(p)（与单位抛物面一起）对平面 $z = 0$ 上各点到 p 的距离进行了“编码”。（由于对任何点 q 都有 $\text{dist}(p, q)^2 \geq 0$，而且 $p' \in h(p)$，这就说明 h(p)必然与 $\mathcal{U}$ 相切于点 p'。）

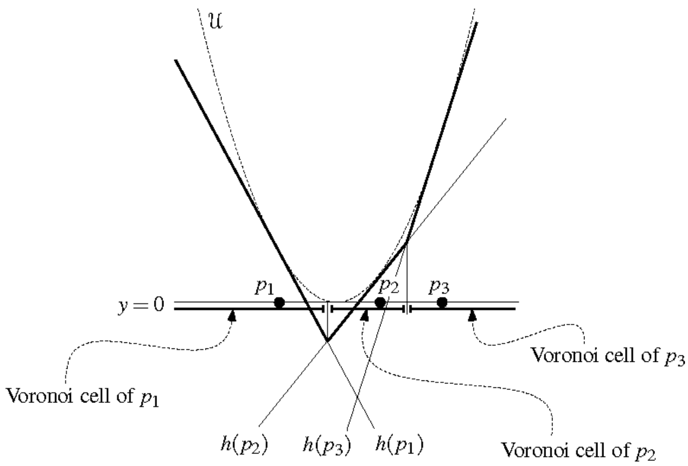


图11-17 Voronoi图与上包络的对应关系

平面h(p)对其它各点到p的距离进行了编码，这一事实将导出Voronoi图与上包络之间的相互对应关系。下面对此做一解释。任取一个平面点集P，我们将该平面假想为三维空间中的平面z = 0。考虑平面集合H := {h(p) | p ∈ P}，令$\mathcal{UE}$(H)为由其中所有平面确定的上包络。我们声称，$\mathcal{UE}$(H)在平面z = 0上的投影，恰好就是P的Voronoi图。图 11-17 说明这一性质，只不过空间降低了一个维度：直线y = 0上任意一组点p_i的Voronoi图，就是由所有直线$h(p_i)$确定的上包络的投影。

〖定理 11.8〗

设 P 为三维空间中的任一点集，其中所有点都来自平面 z = 0。设 H 为由所有点 p ∈ P（按照上述定义）的对偶平面 h(p)所组成的集合。则 $\mathcal{UE}$(H)到平面 z = 0 上的投影，就是 P 的 Voronoi 图。

〖证明〗

为证明该定理，需要说明：任一点 p ∈ P 的 Voronoi 单元，正好就是平面 h(p)为 $\mathcal{UE}$(H)贡献的那张小平面的投影。在平面 z = 0 上，在 p 所对应 Voronoi 单元中任取一点 q。于是，对任何 r ∈ P\{p}，都有 dist(q, p) < dist(q, r)。需要证明的是：穿过 q 的那条垂线与 $\mathcal{UE}$(H)的交点，必落在平面 h(p)上。我们记得，对于任一点 r ∈ P，平面 h(r)都会与穿过 q 的垂线相交于点 $q(r) := (q_x, q_y, q_x^2 + q_y^2 - \text{dist}(q, r)^2)$。既然在 P 内各点中，点 p 到 q 的距离最近，故 q(p)必为最高的交点。因此，正如我们所声称的，穿过 q 的垂线与 $\mathcal{UE}$(H)的交点必然落在平面 h(p)上。 ❑

根据这一定理，为了构造平面Voronoi图，只需构造出三维空间中某一组平面的上包络。根据 习题 11.10 的结论（同时参照上一节），三维空间中一组平面H的上包络，与H^*中各点的下凸包之间相互对应，因此（为求解这类问题）可以直接采用算法CONVEXHULL。

H^*的下凸包也同样有其几何意义：它到平面 z = 0 上的投影，就是 P 的 Delaunay 三角剖分（Delaunay triangulation）——对此，你应该不会感到奇怪。

11.6 注释及评论

早期的凸包算法只能处理平面点集——关于这些算法的讨论，请参见第 1 章的“注释及评论”一节。在三维空间中构造凸包，难度要大很多。第一个此类算法，是Chand和Kapur[84]提出的“礼品包扎”（gift wrapping）算法。该算法用一张平面不断地围绕已知的边 “旋转”，从而逐一找出（构成凸包的）所有小平面；当重新回到起点时，算法终止。对于由f张小平面组成的一个凸包，该算法的运行时间为$O(nf)$——最坏情况下为$O(n^2)$。第一个运行时间不超过$O(n\log n)$的算法，是由Preparata和Hong[322][323]提出的，属于分治式算法。早期递增式算法的运行时间为$O(n^2)$[344][223]。这里所介绍的随机版本，源自Clarkson和Shor[133]。该版本需要$O(n\log n)$空间；而最初的那篇论文还通过一个简单方法，将空间复杂度改进至线性量级。本书中在此处首次使用的冲突图概念，也来自于该篇论文。然而，这里的分析方法却来自于Mulmuley[290]。

本章的注意力集中在三维空间，此时凸包本身的复杂度依然是线性的。所谓的上界定理（upper bound theorem）指出：在 d-维空间中，由 n 个点确定的凸包（就对偶空间而言，即 n 个半空间的公共交集）的组合复杂度（combinatorial complexity）在最坏情况下可以达到$\Theta(n^{\lfloor d/2\rfloor})$。（利用欧拉公式，我们已经证明了 d = 3 的情况。）

本章所介绍的算法，可以推广至更高维的空间，而且在最坏情况下是最优的——其期望运行时间为$\Theta(n^{\lfloor d/2\rfloor})$。有趣的是，对于奇数维空间，已知最好的确定性凸包算法，就是在该算法的基础上，通过（非常复杂的）非随机化转换（derandomization）而设计出来的 [97]。在高于三维的空间中，既然凸包本身的复杂度是超线性的，所以输出敏感的算法将会很有用处。就在$\mathbb{R}^d$中构造凸包的问题而言，在已知的各种输出敏感的算法中，最好的一个是由Chan[82]提出的。该算法的运行时间为$O(n\log k + (nk)^{1-1/(\lfloor d/2\rfloor+1)}\log^{O(1)} n)$，其中k为凸包的实际复杂度。如果读者希望了解高维空间中多胞体的数学性质，可以参考Grunbaum的专著 [194]（它是多胞体理论的经典）；也可以参考Ziegler的专著 [399]，该书就（多胞体的）组合性质进行了论述。

我们在第 11.5 节中已经看到，平面点集的Voronoi图，就是三维空间中某组特定平面的上包络的投影。对于高维空间，类似的结论依然成立：$\mathbb{R}^d$中任一点集的Voronoi图，就是$\mathbb{R}^{d+1}$ 中某组特定超平面的上包络的投影。然而反过来，任意一组（超）平面的上包络的投影，并不见得必然是某个点集的Voronoi图。不过有趣的是，每一个上包络的投影，的确都是一幅所谓的能量图（power diagram）[25]——这种图是Voronoi图的一个推广，其中的基点可以是（超）球，而不再是只限于点。

11.7 习题

习题 11.1　第 1 章曾经将点集P的凸包定义为“包含这些点的所有凸集的公共交集”。本章又给

出了另一种定义——所谓P的凸包，就是“由P中各点的所有凸组合构成的集合”。试证明：这两种定义是等价的（也就是说，你需要证明：q是P中若干点的凸组合，当且仅当q属于以P为子集的任一凸集）。

习题 11.2 试证明：在最坏情况下，算法 CONVEXHULL 的运行时间为 $O(n^3)$；而且，的确存在某些点集，在采用了不当的随机排列时，算法需要运行$\Theta(n^3)$时间。

习题 11.3 试描述一个随机增量式算法，构造平面上任意 n 个点的凸包。你还需要说明对退化情况的处理方法。试对该算法的期望运行时间做一分析。

习题 11.4 在很多的实际应用中，在包含 n 个点的集合 P 内，只有很少比例的点是（其凸包边界上的）极点。在这种情况下，构成 P 的凸包的顶点数目要远远少于 n。事实上，利用这一性质，可以使我们的算法 CONVEXHULL 运行得快于$\Theta(n\log n)$。
比如，假定在规模为 r 的随机采样点集 P 中，极点数目的期望值为 $O(r^\alpha)$，其中常数$\alpha < 1$（如果 P 中各点均匀地分布于一个球体的内部，这一条件就将成立）。试证明：在这一条件下，算法的运行时间为 $O(n)$。

习题 11.5 对于由三维空间中任意 n 个点组成的一个集合 P，可以这样来构造 P 的凸包：将一张平面围绕凸包上的各条边不断“旋转”，并在此过程中逐一找出构成凸包的各张小平面。试按照这一思路，详细给出一个算法，并对其运行时间做一分析。

习题 11.6 试描述一种数据结构，借助该结构能够通过比较判断出，任一给定的待查询点（query point）q是否落在$\mathbb{R}^3$中的某个凸多胞体之内。（提示：利用第 6 章中的结论。）

习题 11.7 所谓简单多胞体（simple polytope），就是三维空间中的一个区域，其边界由多个平面多边形围成，而且它在拓扑上等价于一个球（尽管它本身不见得一定是凸的）。试说明，对于三维空间中由 n 个顶点确定的一个简单多胞体，如何才能在 $O(n)$时间内判断出某个点是否落在其内部。

习题 11.8 试描述一个随机增量式算法，用以构造一组半平面公共交集。试对其期望运行时间做一分析。你的算法需要维护当前半平面集的公共交集。为了确定新半平面的插入位置，还需要维护一幅冲突图，以记录当前交集的各顶点与待插入的各半平面之间的冲突。

习题 11.9 试描述一个随机增量式算法，用以在三维空间中构造一组半空间公共交集。试对其期望运行时间做一分析。与上题类似地，你也需要维护一幅冲突图。

习题 11.10 在本题中，你需要详细地设计出一个对偶变换。对于$\mathbb{R}^3$中的任一点$p := (p_x, p_y, p_z)$，平面$z = p_x x + p_y y - p_z$记作p^*。对于非垂直的平面h，令h^*为满足$(h^*)^* = h$的点。试参照第 11.4 节中针对平面情况所做的相关定义，给出三维点集P的上凸包$\mathcal{UH}(P)$的定义，以及三维空间中平面集H的下包络$\mathcal{LE}(H)$的定义。
试证明下列性质：

- 点p落在平面h上，当且仅当点h^*落在平面p^*上。
- 点p落在平面h上方，当且仅当点h^*落在平面p^*上方。

- 点$p \in P$是$\mathcal{UH}(P)$的一个顶点，当且仅当平面p^*为$\mathcal{LE}(P^*)$贡献一张小平面。
- 线段$\overline{pq}$是$\mathcal{UH}(P)$的一条边，当且仅当平面p^*和q^*的交线为$\mathcal{LE}(P^*)$贡献一条边。
- 点$p_1, p_2, \ldots, p_k$为$\mathcal{UH}(P)$上某张小平面f的顶点，当且仅当平面$p_1^*, p_2^*, \ldots, p_k^*$各自为$\mathcal{LE}(P^*)$贡献的小平面拥有一个公共的顶点。

12

空间二分：画家算法

相对于他们的前辈们，当代的飞行员已今非昔比，他们最初的飞行经验，不是来自于实际的空中飞行，而是借助地面上的飞行模拟器。对航空公司来说，这更加经济；而对飞行员本人来说，这也更加安全；另外，这也更利于环保。只有在模拟器上“飞行”很长时间之后，飞行员才会被允许接触真正飞机上的操纵杆。为了让飞行员忘记自己只不过是坐在一台飞行模拟器上，好的模拟器必须能够完成多种不同的任务。其中一项重要的任务就是可视化（visualization）——让飞行员看见自己下方的景物，或者她们即将降落的跑道。这涉及到对场景的造型（modeling）以及对模型的绘制（rendering）。为了绘制某一场景，对于屏幕上的每个像素（pixel），都必须确定在该像素处哪个

物体才是可见的——即所谓的隐藏面消除（hidden surface removal）。此外，还需要进行光照计算（shading）——亦即，计算出可见物体朝视点方向发射的光线强度。后一项计算可以大大提高图像的真实感，却十分耗时——这不仅要计算出有多少光能够（直接从光源或间接地通过其它物体的反射）到达物体，而且还要考虑到光与物体表面的相互作用，这样才能计算出有多少光朝着视点反射出去。在飞行模拟中，绘制计算必须实时进行，因此来不及进行精确的光照计算。于是，通常采用的都是一种快速而简单的光照计算方法，而隐藏面消除则成为影响绘制时间的一个重要因素。

z-缓冲算法（z-buffer algorithm）就是一种非常简明的隐藏面消除法，其过程如下。首先，对场景进行变换，使得视线方向为正的 z-方向。接下来，按照任意次序对场景中的物体进行扫描转换（scan conversion）。所谓的扫描转换，就是确定被该物体的投影所覆盖的那些像素——只有在这些位置，该物体才有可能是可见的。已经经过处理得到的有关各物体的信息，被该算法存放在两个缓冲（buffer）中——一个称作帧缓冲（frame buffer），另一个称作深度缓冲（z-buffer）。帧缓冲中存储了与每个像素当前可见的物体的（光）强度。所谓“当前可见的物体”，就是在当前已经处理过的各物体中，与某像素可见的物体。深度缓冲中存储了当前与各个像素可见物体的 z-坐标（更准确地，存储的是物体上与像素可见的那一点处的 z-坐标）。现在，假定在对某物体进行扫描转换的过程中，正在处理某一像素。若此物体在该像素处的 z-坐标比当前存储在深度缓冲中的 z-坐标更小，则说明该物体位于当前可见物体的前方。于是，就需要将这一新物体的（光）强度值存入帧缓冲，同时用新的 z-坐标更新深度缓冲。反之，若物体在该像素处的 z-坐标比深度缓冲中当前的 z-坐标更大，则新的这一物体（在此像素处）将不可见，因此帧缓冲和深度缓冲都应该保持原样。z-缓冲算法可以很容易地借助硬件直接实现，因而实际的运行速度很快。正因为此，它成为最常用的隐藏面消除方法。尽管如此，该算法仍有不足之处：深度缓冲需要额外占用大量的存储空间；而且，对于每个物体，被其覆盖的每一个像素都需要进行 z-坐标的测试比较。而画家算法（the painter's algorithm）则可以避免这些计算。该算法首先按照各物体到视点的距离，对其进行排序。然后，从距离视点最远的物体开始，按照所谓的“深度顺序”（depth order），对物体进行扫描转换。在对物体进行扫描转换的过程中，我们不需要对其 z-坐标做任何的测试比较，即可将其（光）强度值直接存入帧缓冲。即使原先帧缓冲中该位置已经存有数值，也可以尽管覆盖。

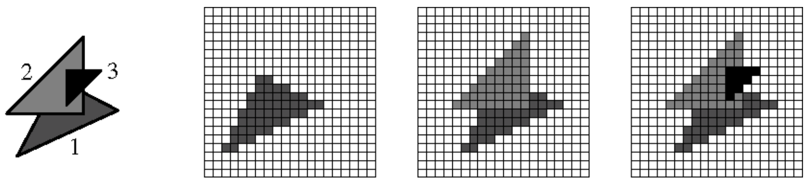


图12-1 画家算法的执行过程

如 图 12-1 所示，就是该算法对一个包含三个三角形的场景进行处理的过程。在该图的左侧，按照扫描转换的次序对各三角形做了编号。右侧所显示的，是依次对第一、第二和第三个三角形完成扫描

转换之后所得到的图像。该算法是正确的，因为我们是按照从后往前的次序对各物体进行扫描转换——这样，对于任一像素，最后写入该像素（在帧缓冲中）对应位置的物体，与视点的距离总是最近的，从而保证了能够得到场景的正确结果。这一算法之所以得名，是由于其过程类似于画家们作画的方式——不断将新的一层颜料涂在原有的一层上面。

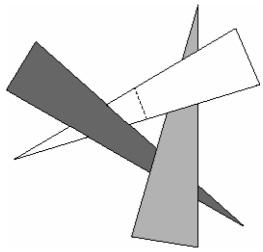


图12-2 多个物体循环覆盖

为了顺利地应用这一方法，必须能够快速地对所有物体进行排序。不幸的是，这并不容易做到。更糟糕的是，并不总是能够按照深度确定一个次序——如 图 12-2 所示，按照“X处于Y前方”的关系，物体之间可能会构成一个循环。一旦出现这种循环覆盖（cyclic overlap）的情况，无论按照何种次序（运行画家算法），都不可能绘制出场景的正确结果。在这种情况下，只能对一个或多个物体进行切分，以切断这种循环关系——其前提是：在分割出来的各块之间，的确存在一个深度次序。以三个三角形为例，无论它们如何构成一个循环，我们总是能够将其中的某一个切分为一个三角形和一个四边形，从而保证在这样的四个物体之间存在一个正确的显示次序。找出需要切分的物体、确定对它们进行切分的位置以及对切分后得到的碎块进行（深度）排序，是一个代价高昂的计算过程。另外，由于深度次序取决于视点的位置，所以在每次视点移动之后，都必须重新进行计算。如果像飞行模拟器这种环境那样，要求以实时的速度运行画家算法，我们就需要对场景进行某种预处理，从而使得对任何一个视点，都能够很快地计算出正确的显示次序。有一种优雅的数据结构可以使之成为可能——这就是空间二分树（binary space partition tree – BSPT），或者简称BSP树。

12.1 BSP 树的定义

为了对BSP树有所体会，请看 图 12-3。图中是平面上一组物体所对应的一个空间二分（binary space partition），以及与之对应的一棵树。正如你所看到的，所谓的空间二分，就是递归地用直线对平面进行分割——第一次，是通过l_1 分割整个平面；接下来，用l_2 分割l_1 上方的半平面，再用l_3 分割l_1 下方的半平面；如此继续。分割线不仅会分割平面，同时也会将某些物体分割成多块碎片。这种分割将一直持续下去，直到每个子区域的内部只包含一个碎片。这样的一个过程，可以很自然地表示为一棵二叉树。在这棵树中，每匹叶子分别对应于最终所得子区域划分（subdividsion）中的一张面；而落在这张面中的那个物体碎片，将存储在该叶子中。每个内部节点则分别对应于一条分割

线；这条线就存储在该节点中。如果场景中包含一维的物体（比如线段），则可能有多个这种物体被包含在某一条分割线中——此时，对应的内部节点需要存储一个列表，以记录这些物体。

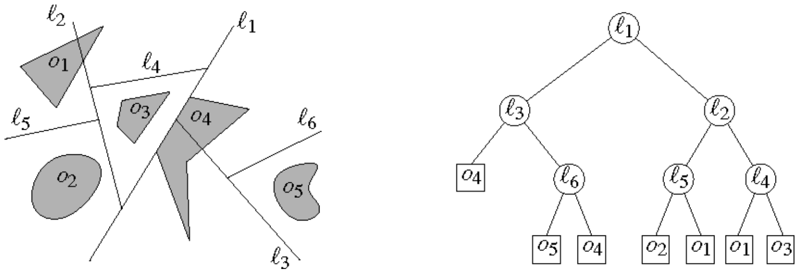


图12-3 空间二分及其对应的树形表示

任一超平面 $h : a_1x_1 + a_2x_2 + \dots + a_dx_d + a_{d+1} = 0$ 都确定了两个开的半空间，我们分别用 h^+和 h^-来表示正、负半空间。即

$$h^+ \ := \ \{(x_1, x_2, \dots, x_d) \mid a_1x_1 + a_2x_2 + \dots + a_dx_d + a_{d+1} > 0\}$$

和

$$h^- \ := \ \{(x_1, x_2, \dots, x_d) \mid a_1x_1 + a_2x_2 + \dots + a_dx_d + a_{d+1} < 0\}$$

这样，对于 d-维空间中任意 n 个物体所组成的集合 S，与之对应的 BSP 树（或者简称 BSP 树）可以定义为一棵具有如下性质的二叉树 $\mathcal{T}$：

- 若 $\text{card}(S) \le 1$，则 $\mathcal{T}$ 是一匹叶子；S 中的物体碎片（如果存在的话）被显式地存储在这匹叶子中。若将这匹叶子记作 v，则存储于这匹叶子处的集合（可能是空集）被记作 S(v)。
- 若 $\text{card}(S) > 1$，则 $\mathcal{T}$ 的根节点 v 存储的是一张超平面 h_v，以及由那些完全落在 h_v 上（≤d-1 维）的物体所组成的集合 S(v)。v 的左孩子为一棵 BSP 子树 $\mathcal{T}^-$的根，这棵树对应于集合 $S^- := \{h_v^- \cap s \mid s \in S\}$；v 的右孩子为另一棵 BSP 子树 $\mathcal{T}^+$的根，这棵树对应于集合 $S^+ := \{h_v^+ \cap s \mid s \in S\}$。

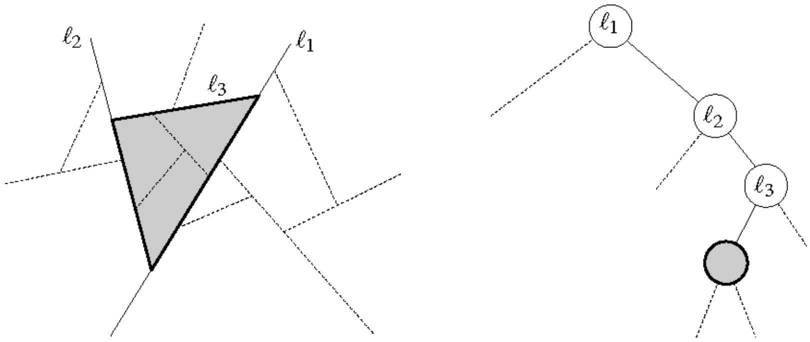


图12-4 节点与子区域之间的对应关系

BSP 树的规模，定义为树中所有节点 v 所对应集合 S(v)的总体规模。换而言之，一棵 BSP 树的

规模就是所生成物体的碎片总数。若 BSP 中没有无用的分割线——即分割空子空间的分割线——则相对于 BSP 树的规模，树中节点的数目至多成线性正比关系。严格地说，根据 BSP 树的规模，并不能确定其所需的存储空间大小——因为，由此并不能知道单块碎片所占用的存储空间。尽管如此，在对同一组物体的不同 BSP 树的质量进行比较时，如上定义的 BSP 树规模，还算是一个不错的标准。

BSP将导出一个子区域划分，而BSP树中的叶子则分别表示其中的各张面。更一般地说，对于BSP树T中的每一个节点，我们都可以找出一个（与之对应的）凸子区域——这个子区域，就是半空间$h_{\mu}^{\Diamond}$的交集（其中，μ为v的祖先。若v来自μ的左子树，则◊ = -；若来自右子树，则◊ = +）。与T的根节点相对应的子区域，就是整个空间。图 12-4 就此给出了说明——其中，灰色节点对应于灰色的子区域$l_1^+ \cap l_2^+ \cap l_3^-$。

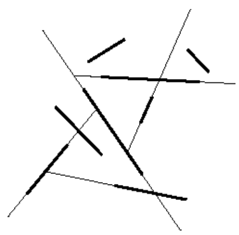


图12-5 自动划分

BSP可用任意超平面进行分割。然而出于计算方便的考虑，可以对允许使用的分割超平面做些限制。一种常见的限制如下。假设我们希望针对平面上的一组线段，构造出与之对应的BSP。这种情况下，一类显而易见的候选分割线，就是这些输入线段所在的直线。如 图 12-5 所示，要是完全利用这类分割线，所生成的BSP就被称作一个自动划分（auto-partition）。如果是三维空间中的一组平面多边形，那么此时所谓的自动划分，就是一个只允许采用各输入多边形所在平面进行分割而得到的BSP。乍看起来，自动划分所受的限制很强。然而，尽管通过自动划分不见得总是能够得到规模最小的BSP树，但是正如我们即将看到的，如此生成的BSP树，规模还是相当小的。

12.2 BSP 树及画家算法

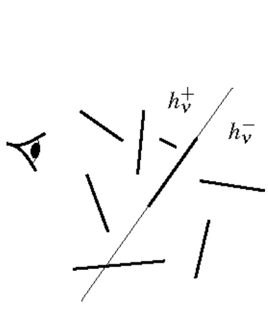


图12-6 相对于上方的视点，下方的物体不会遮挡住上方的物体

假定已经针对三维空间中的一组物体S，构造出了一棵BST树T。应该如何利用T来得到我们所需

要的深度次序，进而通过画家算法来显示集合S呢？设视点为p_{view}，并假设相对于$\mathcal{T}$根节点所存储的那张超平面，p_{view}位于其上方。于是很显然，位于分隔平面下方的任何物体，都不可能遮挡住位于上方的任何物体（如 图 12-6 所示）。因此，只要我们首先显示出子树$\mathcal{T}^-$中的所有物体（更准确地说，应该是物体的碎片），再显示出子树$\mathcal{T}^+$中的物体，就不致会出现问题。至于这两棵子树$\mathcal{T}^-$和$\mathcal{T}^+$内部各物体碎片之间的次序，则可以同样地按照这种方式，递归计算出来。

这一过程，可以总结为下列算法：

```
算法 PAINTERS 算法(T, p_view)
1.  设 v 为 T 的根节点
2.  if (v 是叶子)
3.     then 对 S(v)中的所有物体碎片进行扫描转换
4.     else if (p_view ∈ h_v^+)
5.             then PAINTERS 算法(T^-, p_view)
6.                  对 S(v)中的所有物体碎片进行扫描转换
7.                  PAINTERS 算法(T^+, p_view)
8.             else if (p_view ∈ h_v^-)
9.                     then PAINTERS 算法(T^+, p_view)
10.                         对 S(v)中的所有物体碎片进行扫描转换
11.                         PAINTERS 算法(T^-, p_view)
12.                    else (* p_view ∈ h_v *)
13.                         PAINTERS 算法(T^+, p_view)
14.                         PAINTERS 算法(T^-, p_view)
```

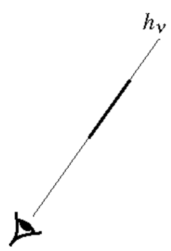


图12-7 视点恰好落在分割平面上

请注意，如 图 12-7 所示，当p_{view}正好落在分割平面h_v上时，我们并不画出S(v)中的任何多边形——因为多边形都是二维平面物体，所以从与其共面的任何一个点来看，它们都不是可见的。

该算法——以及任何采用 BSP 树的此类算法——的效率，主要取决于 BSP 树的规模。因此，

我们必须仔细选用分割平面，从而使得各物体被分割成的碎片最少。何种策略才能生成规模更小的 BSP 树呢？在开始讨论这一问题之前，必须首先明确，什么类型的物体才是允许的。之所以对 BSP 树感兴趣，是因为我们希望针对飞行模拟器这一应用，找到隐藏面消除的一种快速实现方法。既然最关注的是速度，就应该使场景中的物体尽可能简单——这就是说，我们不能使用曲面，而只能将每个物体都表示为多面体模型。这里假定，多面体的每一小平面都已经做过三角剖分。因此，我们的任务就是：针对三维空间中给定的一组三角形，构造出一棵规模尽可能小的 BSP 树。

12.3 构造 BSP 树

在着手解决某个三维问题的时候，通常可以采用这样一个不错的方法：首先对该问题的平面简化版本做一研究，借此获得对问题的认识。本节也将采用这一方法。

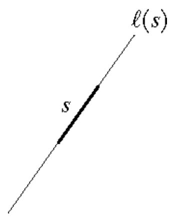


图12-8 沿输入线段进行分割

设S为平面上一组共n条互不相交的线段。我们首先将注意力放在自动划分上——如 图 12-8 所示，我们暂且要求，只有S中各线段所在的直线，才能做为分割线。下面这个构造BSP的递归算法，是不证自明的。任一线段s所在的直线，记作l(s)。

算法 2DBSP(S)

输入：一组线段 $S = \{s_1, s_2, \ldots, s_n\}$

输出：S 的一棵 BSP 树

1. **if** (card(S) ≤ 1)
2. **then** 直接生成一棵只含单匹叶子的树 $\mathcal{T}$，集合 S 显式地存储于其中
3. **return** $\mathcal{T}$
4. **else** (* 将 $l(s_1)$做为一条分割线 *)
5. $S^+ \leftarrow \{s \cap l(s_1)^+ \mid s \in S\}$； $\mathcal{T}^+ \leftarrow$ 2DBSP(S^+)
6. $S^- \leftarrow \{s \cap l(s_1)^- \mid s \in S\}$； $\mathcal{T}^- \leftarrow$ 2DBSP(S^-)
7. 生成一棵 BSP 树 $\mathcal{T}$
（其根节点为 v，左子树为 $\mathcal{T}^-$，右子树为 $\mathcal{T}^+$，
而且 $S(v) = \{s \in S \mid s \subset l(s_1)\}$）
8. **return** $\mathcal{T}$

显然，该算法能够构造出任一集合 S 对应的 BSP 树。问题在于，这棵树是否足够小？在选用线段做分割时，或许应该做更多计算以选出更好的线段，而不是如这里做法——直接选用第一条线段 s_1。马上能够想到的一个办法是：选用使得 l(s)与尽可能少的线段相交的那条线段 $s \in S$。然而，这种策略过于贪心了——对于某些由线段组成的构形，该策略是行不通的。此外，为了找出这样的线段，也需花费大量时间。那么，还有其它的办法吗？或许你已经猜出来了——正如前几章的做法一样，在难以做出选择的时候，干脆随机地进行选择。针对当前问题的做法是，随机地挑出一条线段，用作分割线。稍后我们将会看到，如此构造出来的 BSP，其期望规模的确非常小。

为了实现这一构思，在构造过程开始之前，需要把所有线段的次序随机打乱。

算法 2DRANDOMBSP(S)
1. 生成集合 S 中各线段的一个随机排列 $S' = s_1, \ldots, s_n$
2. $\mathcal{T} \leftarrow$ 2DBSP(S')
3. **return** $\mathcal{T}$

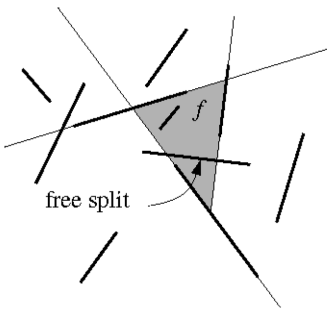


图12-9 直接采用免费的分割

在对这一随机算法进行分析之前，我们需要注意到，有一些简单的优化可以直接进行。假设已经选取了前面若干条分割线。由于这些直线的引入，在平面上导出了一个子区域划分，其中的各张面分别对应于我们正在构造的BSP树中的各个节点。任取其中一张面f。有些线段可能会完全跨越f。如 图 12-9 所示，如果选取这样的一条线段来对f进行分割，f内部的其它线段将不会因此而被分割开来；而且，在此后的计算中我们可以不再考虑这条线段。看到这类免费的分割（free split）而不加以利用，简直就是傻瓜。因此改进后的策略是，免费的分割一旦可行，就立即采用；只有在不存在这类分割方案时，才进行随机的分割。为了实现这一优化，我们必须能够判断出来，一条线段到底是不是一个免费的分割。为此，需要为每条线段分别维护两个布尔型变量，通过它们可以知道，（每条线段的）左、右端点是否落在某条已经采用过的分割线的同侧。一旦这两个变量同时为真，对应的线段就是一个免费的分割。

现在，对算法 2DRANDOMBSP 的性能做一分析。为了简化分析，只分析未采用免费分割的版本。（实际上就渐进复杂度而言，即使采用了免费分割，也不会有什么改进。）

首先来分析BSP树的规模——即由算法生成的碎片总数。当然，它主要取决于第一行所选定的排列次序——有些排列将导致更小的BSP树，另一些则更大。做为例子，请考察如 图 12-10 所示的一组共 3 条线段。如果按如(a)所示的次序处理这些线段，将得到 5 块碎片。然而，如果按如(b)所示的另一次序，则只生成 3 块碎片。可见，采用不同的次序将得到不同规模的BSP。因此，只能分析BSP树的期望规模（expected size）——即全部n!种排列（所得到BSP树）的平均规模。

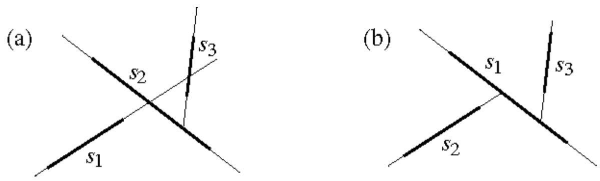


图12-10 不同的次序导致不同BSP

〖引理 12.1〗

算法 2DRandomBsp 所生成碎片的期望数目为 $O(n\log n)$。

〖证明〗

任取 S 中一条固定的线段 s_i。在算法将 $l(s_i)$做为下一条分割线引入时，在其它（已经引入的）的线段中，按照期望估计会有多少条与 $l(s_i)$相交呢？我们将对此做一分析。

在引入$l(s_i)$时，另一线段s_j即使就相对位置而言的确与该直线相交，也未见得一定会被$l(s_i)$分割。什么时候才会被分割呢？由 图 12-10 可以看出，这要取决于“夹在”s_i和s_j之间、与$l(s_i)$相交的那些线段。因此，只要对这些线段进行检查，就可以做出判断。具体地说，如果的确存在这样一条线段，而且其所在的直线已经先于$l(s_i)$被用作分割线，那么s_j就会被这条直线“庇护”起来，从而不致于被$l(s_i)$切分。图 12-10 中的图(b)就是这种情况——由于线段s_1 的“庇护”，s_3 没有被$l(s_2)$切分（尽管s3 的确与$l(s_2)$相交）。出于这种考虑，我们将为每一条线段，定义其相对于某条固定线段s_i的距离：

$$\mathrm{dist}_{s_i}(s_j) = \begin{cases} \text{介于}s_i\text{与}s_j\text{之间、与}l(s_i)\text{相交线段的条数} & (\text{若}l(s_i)\text{与}s_j\text{相交}) \\ +\infty & (\text{否则}) \end{cases}$$

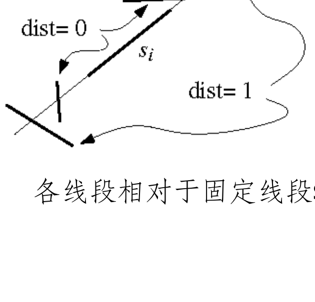


图12-11 各线段相对于固定线段s_i的距离

如图 12-11 所示，就有限距离而言，拥有同一距离的线段最多不会超过两条——在s_i的两翼，各有一条。

设 $k := \mathrm{dist}_{s_i}(s_j)$，而介于 s_i 和 s_j 之间的这 k 条线段（依次）为 $s_{j_1}, s_{j_2}, \ldots, s_{j_k}$。在将 $\ell(s_i)$做为最新的分割线引入的时候，它将 s_j 切分开来的概率是多大呢？要想发生切分，第一个必要条件就是，在随机序列中 s_i 必须排在 s_j 之前；此外，s_i 还必须排在介于 s_i 和 s_j 之间的所有线段之前——否则，其中任何一条线段都会把 s_j“庇护”起来，使之免于被 s_i 切分。换而言之，在下标集合$\{i, j, j_1, \ldots, j_k\}$当中，i 必须是最小的。既然各线段的次序是随机的，这就意味着

$$\Pr[\ell(s_i)\text{切分 } s_j] \leq \frac{1}{\mathrm{dist}_{s_i}(s_j) + 2}$$

需要注意的是，某些本身并未被 $\ell(s_i)$切分的线段，它们的延长线同样有可能会将 s_j“庇护”起来。上式之所以不是等号，原因正在于此。

现在，我们就可以界定由于 s_i 的引入而造成切分的期望数目如下：

$$E[s_i\text{ 产生的切分数目}]$$

$$\leq \sum_{j\neq i}\frac{1}{\mathrm{dist}_{s_i}(s_j) + 2}$$

$$\leq 2 \cdot \sum_{k=0}^{n-2}\frac{1}{k+2}$$

$$\leq 2 \cdot \ln n$$

由期望的线性律可知：所有线段所产生切分的总期望数目不会超过 $2n\cdot\ln n$。我们开始于 n 条线段，而每一切分都使碎片数目加一，故最终碎片的期望数目不会超过 $n + 2n\ln n$. ❑

我们已经证明，由算法 2DRANDOMBSP 生成的 BSP，期望规模为 $n + 2n\ln n$。由此也就证明了，对于包含 n 条线段的任一集合，都*必然存在*一个规模为 $n + 2n\ln n$ 的 BSP。而且，在所有的排列次序中，至少有一半排列次序所生成的 BSP 规模不超过 $n + 4n\ln n$。可以利用这一点，找到规模如此小的一个 BSP——在算法 2DRANDOMBSP 启动之后，我们对树的规模进行监视；一旦中途超过这一界限，就换成另一个随机排列次序，重新运行这一算法。我们需要尝试的次数，期望值为 2。

我们已经对算法 2DRANDOMBSP 所生成 BSP 的规模进行了分析。那么，运行时间呢？同样地，这也取决于我们所采用的随机排列次序。因此，也只能考虑期望的运行时间。可以在线性时间内计算出一个随机排列。如果忽略递归调用所消耗的时间，算法 2DBSP 所需的时间将线性正比于 S 中的碎片数目。这个数目绝不可能超过 n——实际上，随着递归深度的增加，这个数只会越来越小。最

后，递归调用的次数显然不会超过生成的碎片数目，也就是 $O(n\log n)$。因此，总的构造时间为 $O(n^2\log n)$，这样就得到如下结论：

〖定理 12.2〗

可以在 $O(n^2\log n)$期望时间内，构造出一个规模为 $O(n\log n)$的 BSP。

由 2DRANDOMBSP构造出的BSP，其期望规模还算不错；尽管如此，该算法的运行时间却多少有点令人失望。在很多应用中，这并不会显得多重要——因为在这些环境中的构造过程，可以是离线（off-line）进行的。此外，只有在BSP极不平衡的时候，构造过程才需要平方量级的时间，而这种情况在实践中非常少见。尽管如此，从理论的角度来看，这样的构造时间毕竟不能令人满意。通过一种基于线段树（segment tree）结构的方法（参见第 10 章），可以对此进行改进——我们可以使用某种确定性算法（deterministic algorithm），在$O(n\log n)$时间内构造出一个规模为$O(n\log n)$的BSP。不过，这种方法所得到的将不再是一个自动划分；而且在实践中，它所生成的BSP规模稍嫌庞大。

一个自然会想到的问题就是：2DRANDOMBSP 所生成 BSP 的规模，是否同样可能得到改进？——对于平面上的任何线段集，是否都存在一个规模为 $O(n)$的 BSP？或者反过来，是否存在某些集合，它们的任何 BSP 都具有$\Omega(n\log n)$的规模？截至目前，其答案依然是未知的。

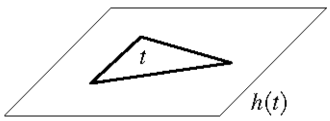


图12-12　　三角形所在的平面

这里针对平面情况所介绍的算法，可以马上推广至三维空间。设S为由$\mathbb{R}^3$ 中任意n个互不相交三角形组成的集合。同样地，将问题范围限制于自动划分——亦即，我们只能使用S中各三角形所在的平面进行分割。如 图 12-12 所示，将任一三角形t所在的平面记作h(t)。

算法　3DBSP(S)

输入：$\mathbb{R}^3$ 中的一组三角形 $S = \{t_1, \ldots, t_n\}$

输出：与 S 对应的一棵 BSP 树

1.　**if** (card(S) ≤ 1)
2.　　**then** 直接生成一棵只含单匹叶子的树$\mathcal{T}$，集合 S 显式地存储于其中
3.　　　**return** $\mathcal{T}$
4.　　**else** (* 将 $h(t_1)$做为一张分割面 *)
5.　　　$S^+ \leftarrow \{t \cap h(t_1)^+ \mid t \in S\}$；$\mathcal{T}^+ \leftarrow$ 3DBSP(S^+)
6.　　　$S^- \leftarrow \{t \cap h(t_1)^- \mid t \in S\}$；$\mathcal{T}^- \leftarrow$ 3DBSP(S^-)
7.　　　生成一棵BSP树$\mathcal{T}$

（其根节点为v，左子树为$\mathcal{T}^-$，右子树为$\mathcal{T}^+$，
而且$S(v) = \{t \in S$[①] $\mid t \subset h(t_1)\}$）。

8. **return** $\mathcal{T}$

如此生成的 BSP，其规模取决于各三角形的次序——某些次序产生的碎片要比另一些次序更多。与平面情况一样，也可以在一开始就将所有的三角形打乱成随机次序，以获得更好的期望规模。在实践中，使用这种方法通常都能得到好的结果。然而，依然有待于从理论上对该算法的期望性能做出分析。因此，接下来的一节将对该算法的另一变种进行分析——尽管在实践中，上述算法的性能可能更好。

12.4 *三维 BSP 树的规模

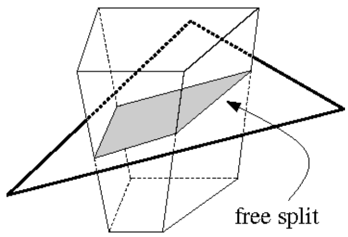


图12-13 三维空间中的免费分割

本节将分析一个在三维空间中构造BSP树的随机算法，该算法与上面所介绍的算法几乎一样——它按照随机的次序来处理各三角形；而且，只要可能，它也会利用免费分割。如 图 12-13 所示，对于这类问题，当S中的某个三角形S将原先的一个单元分割为两个互不连通的子单元时，就会出现一个免费分割。唯一的区别是：在将某张平面h(t)当作分割平面时，需要对所有与该平面相交的单元进行分割，而不仅仅是与t相交的那些单元。（因此，无法简单地通过递归来实现这一算法。）在这样一种例外情况下，我们不要用h(t)对所有单元进行分割：当该分割对某个单元不起实质作用时——即这个单元中的所有三角形都完全落在该平面的同一侧——就不进行分割。

图 12-14 给出了这样一个二维的例子。该图的(a)部分，为利用上一节的算法（依次）对线段s_1、s_2 和s_3 进行处理之后得到的结果。该图的(b)部分，为通过修改后的算法所得到的结果。请注意：在$l(s_1)$下方的子空间内，修改后的算法依然会使用$l(s_2)$进行分割；在$l(s_2)$右边的子空间内，依然用$l(s_3)$进行分割。而在介于$l(s_1)$和$l(s_2)$之间的子空间内，却没有使用$l(s_3)$——这是因为在这里$l(s_3)$的确没有实质的作用。

① 原书此处误作"$t \in T$"。——译者

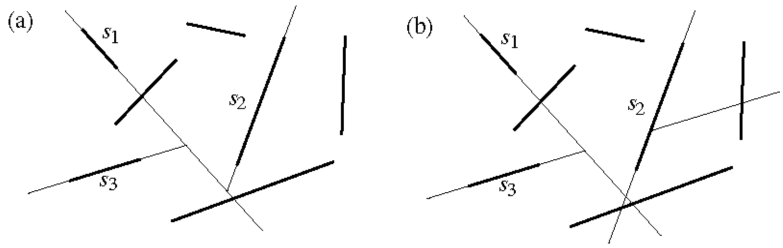


图12-14 原先的算法及修改后的算法

修改后的算法可以总结如下。其中的具体细节，做为习题留给读者。

算法 3DRANDOMBSP2(S)

输入：$\mathbb{R}^3$ 中的一组三角形 $S = \{t_1, t_2, \dots, t_n\}$

输出：对应于 S 的一棵 BSP 树

1. 生成集合 S 的一个随机排列：$t_1, \dots, t_n$
2. **for** $i \leftarrow 1$ **to** n
3. **do** 用 $h(t)$分割所有单元——只要分割能起实质的作用
4. 进行所有的免费分割

关于该算法所生成碎片的期望数目，下面这则引理给出了分析。

〖引理 12.3〗

就全部 n!种可能的排列而言，算法 3DRANDOMBSP2 所生成碎片的期望数目为 $O(n^2)$。

〖证明〗

我们需要确定，任何一个三角形$t_k \in S$最多可能被分割成多少块碎片。对于任何满足$i < k$的三角形t_i，定义$l_i := h(t_i) \cap h(t_k)$。如图 12-15 所示，集合$L := \{l_1, \dots, l_{k-1}\}$由平面上不超过$k$-1条直线组成。其中有些直线与$t_k$相交，有些则不相交。

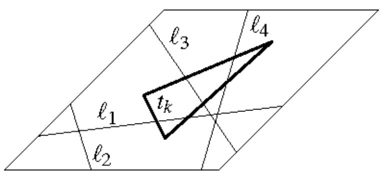


图12-15 $\{l_1, \dots, l_{k-1}\}$与t_k

如果其中某条直线 l_i 与 t_k 相交，就定义 $s_i := l_i \cap t_k$。令 I 为所有这些交（线段）组成的集合。由于使用了免费分割，t_k 可能被分割成的碎片块数，通常不会简单地等于 I 在 t_k 中所导出排列（arrangement）中的面（face）的数目。为了理解这一点，试考察正在处理 t_{k-1} 的时刻。我们假设 l_{k-1} 与 t_k 相交；否则，t_{k-1} 就不可能在 t_k 中分割出任何碎片。考虑由 $I\backslash\{s_{k-1}\}$在 t_k 中导出的排列。在其中，线段 s_{k-1} 可能会与多张面相交。然而，要是有这样的某张面 f 与 t_k 的三条边之

一相关联——这样的 f 被称作内部面（interior face）——那么必然已经沿着 t_k 的这个部分做过了一次免费分割。也就是说，$h(t_{k-1})$只可能引起对外部面（exterior face）——即与 t_k 三条边之一相关联的面——的切分。因此，由 $h(t_{k-1})$在 t_k 中引起的分割数目，将取决于由 I 在 t_k 内导出的排列中所含的外部面；确切地说，分割的数目等于 s_{k-1} 为这些外部面所贡献的边数。（在后面的分析中，很重要的一点是：外部面集合的组成，与$\{t_1, ..., t_{k-1}\}$的处理次序无关。前一节的算法，并不满足这个条件，这正是我们对其进行修改的原因。）那么，这些边的期望数目又是多少呢？要回答这个问题，首先来界定外部面所含边的总数。

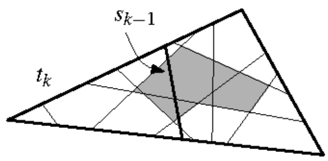


图12-16　在L对应的排列中，需要计数的边必然属于$l(e_1)$、$l(e_2)$或者$l(e_3)$之一所对应的带域

第 8 章引入了带域（zone）的概念——给定在平面上由一组直线构成的排列（arrangement），任何直线l对应的带域由该排列中与l相交的所有面组成。你应该记得，若排列由m条直线构成，则带域的复杂度为$O(m)$。现在，分别设t_k的三条边为e_1、e_2和e_3；对于$i \in \{1, 2, 3\}$，将e_i所在的直线记作$l(e_i)$。如图 12-16 所示，在由集合L在平面$h(t_k)$上导出的排列中，我们所感兴趣的那些边，必然属于$l(e_1)$、$l(e_2)$或者$l(e_3)$之一所对应的带域。因此，外部面所含边的总数为$O(k)$。

如果外部面所含边的总数为 $O(k)$，那么平均落在每条线段上的边为 $O(1)$条。既然 $t_1, ..., t_n$ 是一个随机的排列，$t_1, ..., t_{k-1}$ 必然也是随机的。因此，落在线段 s_{k-1} 上的边的期望数目应为常数，于是由 $h(t_{k-1})$在 t_k 中引起的附加碎片的期望数目应为 $O(1)$。同理可证，从 $h(t_1)$到 $h(t_{k-2})$的每张分割平面，在 t_k 中生成碎片的期望块数都是常数。这就是说，t_k 被切分成碎片的块数，期望值为 $O(k)$。因此，碎片的总数就是

$$O(\sum_{k=1}^{n} k) = O(n^2)$$

❑

由 3DRandomBsp 生成的空间二分，期望的规模为平方量级，由此立即可知，存在一棵平方量级的 BSP 树。

上面所得到的复杂度上界（upper bound），多少有点令人失望。如果需要处理的是 10,000 个三

角形，恐怕你不会愿意使用一棵规模高达平方量级的BSP树。下面这则引理[①]将告诉我们，只要依然限制于自动划分，就不可能指望证明出一个更好的结果。

〖引理 12.4〗

在三维空间中，存在一组共 n 个互不相交的三角形，对它们的任何一个自动划分，规模都至少为$\Omega(n^2)$。

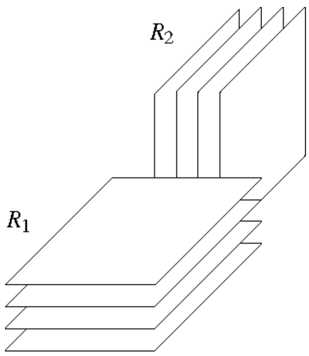


图12-17 自动划分的最坏情况

〖证明〗

考虑如 图 12-17 所示的一组矩形，其中R_1为平行于xy-平面的一组矩形，R_2为平行于yz-平面的另一组矩形。（使用三角形也可以，不过使用矩形更易于理解。）设$n_1 := \text{card}(R_1)$，$n_2 := \text{card}(R_2)$。考察该构形的所有自动划分，令$G(n_1, n_2)$为它们的最小规模。我们断言：$G(n_1, n_2) = (n_1 + 1)(n_2 + 1) - 1$。这可以通过对$n_1 + n_2$进行归纳来证明。对于$G(1, 0)$和$G(0, 1)$，上述断言显然成立。这样，让我们来考虑$n_1 + n_2 > 1$ 的情况。不失一般性地，假定自动划分从R_1中选用了一个矩形r。r所在的平面必然会对R_2 中的所有举行进行切分。接下来，要对经过分割所得到的两个子场景分别进行递归处理，而每个子场景中的情况与原先的构形完全一样。如果将R_1中落在r上方矩形的数目记作m，就得到了：

$$
\begin{aligned}
&G(n_1, n_2) \\
&= 1 + G(m, n_2) + G(n_1 - m - 1, n_2) \\
&= 1 + ((m+1)(n_2+1) - 1) + ((n_1-m)(n_2+1) - 1) \\
&= (n_1 + 1)(n_2 + 1) - 1
\end{aligned}
$$

❑

因此，或许不应该只限制于自动划分。就〖引理 12.4〗证明中的下界（lower bound）而言，限制于自动划分的确不是好主意——我们已经证明，这种划分的规模都不会低于平方量级；而要是在一开始就使用一张与xz-平面平行的平面分割出集合R_1与R_2，则可简明地得到一个线性规模的BSP。

[①] 原书误作"theorem"，应为"lemma"。——译者

然而要是换成如 图 12-18 所示的构形，则即使不做任何限制，也不能生成一个小规模的BSP。

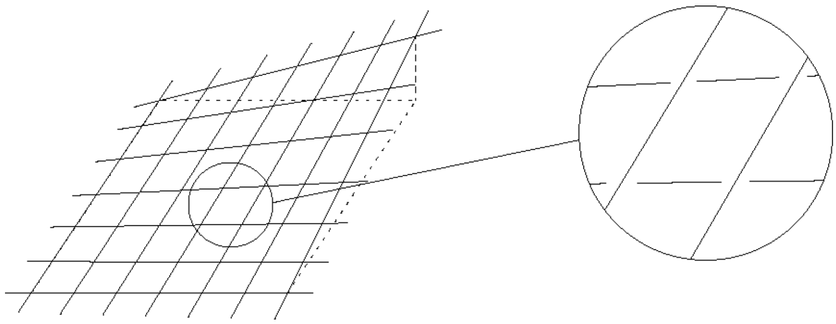


图12-18 一般性下界的构造

这个构形的构造方法如下。从平面上的一个网格开始，这个网格由n/2 条平行于x-轴的直线以及n/2 条平行于y-轴的直线构成。（除直线外，也可以采用狭长的三角形。）略微扭动这些直线，从而得到如 图 12-18 所示的一个构形。在这个构形中，所有直线都（完全）落在所谓的双曲抛物面（hyperbolic paraboloid）上。最后，将与y-轴平行的所有直线向上稍微移动一点，使得任何直线都不再相交。至此，我们所得到的就是如下的一组直线：

$$\{y = i, z = ix \mid 1 \le i \le n/2\} \cup \{x = i, z = iy + \varepsilon \mid 1 \le i \le n/2\}$$

其中，ε为一个很小的正数。只要ε足够小，无论是哪个 BSP，对于每一个格子单元 C，该 BSP 都会在与该 C 紧邻的某个单元中，将围成 C 的四条直线中的至少一条切分开来。（通过初等的方法，即可严格地证明这一事实，不过这一证明既单调乏味，也没有多大益处。其思想是证明：只要适当地扭动各直线，就可以使得任一平面都不可能同时穿过一个格子单元四角处的“缝隙”）。既然总共有平方数量的格子单元，当然就会生成$\Theta(n^2)$ 块碎片。

〖定理 12.5〗

对于 $\mathbb{R}^3$ 中任意一组共 n 个互不相交的三角形，都存在一棵规模为 $O(n^2)$的 BSP 树。此外，的确存在某些构形，其任何 BSP 的规模都至少是$\Omega(n^2)$。

BSP 树规模的平方量级下界，可能会造成一种印象：它们在实践中没有用处。幸运的是，事实并非如此。达到下界的那些构形都是刻意人为的。在许多实际情况中，BSP 树的性能都很好。

12.5 低密度场景的 BSP 树

针对 $\mathbb{R}^3$ 中任意一组共 n 个互不相交的三角形，前一节介绍了 BSP 树的一个构造算法。如此构造出来的 BSP 树，规模总是 $O(n^2)$。我们也曾给出了一组共 n 个三角形的实例，它的任何一棵 BSP 树

的规模都是$\Omega(n^2)$。就这个意义而言，$O(n^2)$这一上界在最坏情况下已是紧的；从理论上看，这一问题至此已告解决。鉴于其平方量级的上界，人们可能会认为 BSP 树在实践中不会有太大用处。幸运的是，实际情况并非如此。在许多的实际应用场合，BSP 树都能大显身手。显然，关于 BSP 树 de 理论分析，与其实际的工作效率并不吻合。

这将令人苦恼：根据理论分析的结论，在实践中功用良好的这一结构居然应该被放弃！问题的原因在于，某些输入的确会导致 BSP 切割出很多个物体，但其它一些输入却允许 BSP 仅切割出很少的物体。前一情况的实例，包括此前为证明下界而做的网格状构造；而后一情况则在实践中屡见不鲜。我们自然希望理论的分析能够反映这一事实——为此，需要针对不同类型的输入估计其各自的上界。也就是说，不能依然仅就输入的规模 n 作分析。需要引入并考虑其它的参数，该参数应当能够刻画不同输入的容易或棘手程度。那么，什么样的输入是“容易的”呢？直观上看，此类输入中的物体应当是相对易于分隔的，而“棘手的”输入中的物体相互之间应当更为紧凑。需要指出的是，物体是否相互靠近，与它们之间的绝对距离无关，而是与物体之间的距离相对于物体本身的尺寸相关——否则，只需简单地放大整个场景，难易程度的判定结论就不是确定的了，这是我们所不希望的。因此，我们按照以下的定义，引入密度（density）这一参数。

将物体o的直径记作diam(o)。对于$\mathbb{R}^d$中的任意一组物体S，其密度为满足以下条件的最小数λ：任意球体B最多与S中满足diam(o)≥diam(B)的λ个物体$o \in S$相交。这一定义如 图 12-19 所示。需要强调的是，这里指的是“任意”球体B：B本身不属于S，但中心位置、半径可以任意取定。

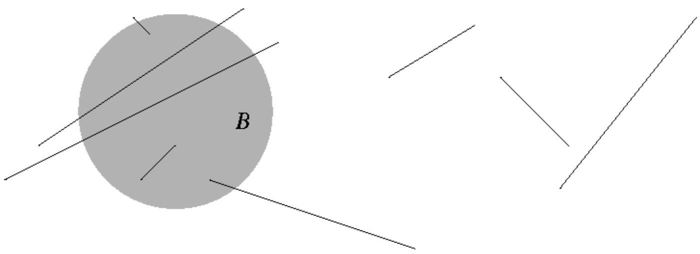


图12-19　这8条线段组成的集合，密度为3。圆盘B尽管与5条线段相交，但其中两条因直径小于diam(B)而不被计入

不难直接构造一组共n个物体，使它们的密度为n——比如任意一组直线。即便限定各物体有界，物体集的密度仍可足够高。比如 图 12-18 中所构造的那组排列成网格的物体，尽管它们都是线段而非直线，其密度依然高达$\Theta(n)$。反过来，有些物体集的密度也可能极低——比如相距超过单位距离的n个单位球，密度为 1。实际上可以证明：任意n个互不相交球体的密度均为$\Theta(1)$，物体其半径相差如何悬殊（习题 12.13）。

现在做一回顾。我们已经定义了一个密度参数，就以下意义，通过该参数来反映场景的难易程度：密度低，则物体相对地更加便于分离；密度高，则意味着某些区域有大量物体聚集。以下试图

证明：密度低——比如说，固定为一个与 n 无关的常数——时，可以得到规模小的 BSP。也许首先想到的是，对前一节中的随机算法做更为细致的分析，并由此证明，那个算法对于低密度的输入可以构造出更小规模的 BSP。然而不幸的是，此方法行不通。因为事实上，即便是对于低密度的输入，那个算法所构造出 BSP 的规模依然可能高达平方量级。换而言之，那个算法并不总是能够充分利用“容易的输入场景”这一条件。因此，必须重新设计算法。

设 S 为 $\mathbb{R}^2$ 中的一组物体（线段、圆盘、三角形等等），S 的密度为λ。（实际上，以下介绍的算法同样适用于 $\mathbb{R}^3$，甚至更高维的空间。为使讲解简明，以下不妨限定在 $\mathbb{R}^3$。）算法的思想是：为每个物体 o∈S 定义少量的一组点——称作哨兵（guard）——使得哨兵的分布反映物体的分布，并进而依靠这些哨兵的辅助来构造 BSP。以下介绍这一思路的具体实现。

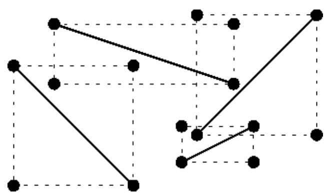


图12-20 物体的包围框及其哨兵

将 o 的包围框（bounding box）——亦即包围 o、与坐标轴平行的最小矩形——记作 bb(o)。于是，可以直接选取 bb(o)的四个顶点作为 o 的哨兵。S 中所有物体的 4n 个哨兵，组成一个复集（multiset），记作 G(S)。（G(S)之所以是一个复集，是因为不同物体的包围框可能顶点重合。此时，我们依然希望这些哨兵以重复的次数计入 G(S)。）就以下意义而言，低密度 S 的 G(S)中的哨兵的确可以反映 S 中物体的分布：S 中与任一正方形 σ 相交的物体，不会超过落在 σ 内部的哨兵数。下面的引理，将对此给出精确的印证。需要指出的是，该引理仅给出了与同一正方形相交的物体的数目上界，而非下界——事实上很有可能出现的情况是，某个正方形包含很多 guard，却与任何物体都不相交。另外，尽管此前是通过圆盘来定义（二维空间中）的密度，但以下引理所提到的哨兵性质，却是相对正方形而言的。

〖引理 12.6〗

平行于坐标轴、内含 G(S)中 k 个哨兵的正方形，至多与 S 中的 k+4λ个物体相交。

〖证明〗

任取平行于坐标轴、内含 G(S)中 k 个哨兵的一个正方形 σ。显然，σ 内的哨兵（亦即包围框的顶点）至多来自 k 个物体。S 中其余的（即四个哨兵均未落在 σ 内部的）物体，构成集合 S'。易见，（作为 S 子集的）S'的密度不超过λ。以下只需证明，S'中至多有 4λ个物体与 σ 相交。

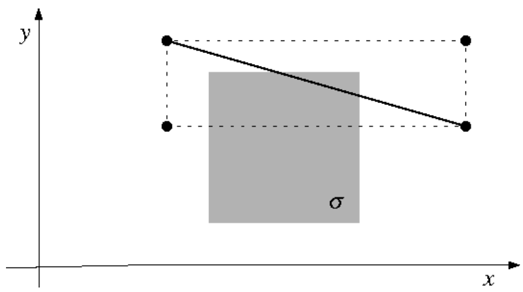


图12-21 正方形σ与一条线段相交，但可能不包含该线段的所有哨兵。果真如此，则该线段的直径不会小于σ的边长

若物体o∈S'与σ相交，则显然bb(o)也必与σ相交。由S'的定义，bb(o)的顶点均不会落在正方形σ内部。于是如 图 12-21 所示，要么bb(o)到x-轴的投影覆盖σ到x-轴的投影，要么bb(o)到y-轴的投影覆盖σ到y-轴的投影，甚至兼而有之。这就意味着，o的直径不会小于σ的边长，即 $daim(o) \geq diam(\sigma)/\sqrt{2}$。若用四个分别以各顶点为圆心、直径为diam(σ)/2 的圆盘$D_1, \ldots, D_4$覆盖σ，则物体o必与其中至少一个圆盘D_i相交。将o计入D_i的名下。于是有：

$$diam(o) \geq diam(\sigma)/\sqrt{2} \geq diam(\sigma)/2 = diam(D_i)$$

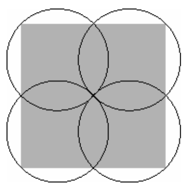


图12-22 用四个直径为diam(σ)/2的圆盘覆盖σ

既然 S'的密度不超过λ，故每个 D_i 至多有λ项入账。因此，σ 至多与 S'中的 4λ个物体相交。加上此前 S'之外的那 k 个物体，可知 σ 至多与 S 中的 k+4λ个物体相交。 ❑

由〖引理 12.6〗，可以导出以下两段式BSP构造算法。这里，令U为包含S中所有物体的一个正方形。

算法的前一阶段，递归地对U做正方形划分，直到每个正方形包含的哨兵不超过一个 [①]。换而言之，也就是构造G(S)[②]的一棵四叉树（quadtree，参见第 14 章）。为将一个正方形分成四个象限，可以先用垂直线一分为二，再用水平线二分为四。如此，如 图 12-23 所示，可由点集G(S)的四叉树划分（quadtree subdivision），导出一棵BSP树（BSP tree）。请注意，有的划分线（比如l_2和l_3）实际上是同一条线；其区别在于它们分别与该直线的哪一段相关，尽管BSP树中的节点并不会记录这

① 重复的哨兵算做一个。——译者。

② 原书误作 G(s)。——译者。

一信息。根据〖引理 12.6〗，该四叉树划分中的每一叶子区域，仅与很少量的物体相交——准确地讲，至多 1+4λ个。算法的后一阶段，将对每块叶子区域做进一步划分，直到所有物体相互分离。这一阶段具体如何实施，完全取决于S中物体的类型。比如，要是所有物体都是线段，则可套用第 12.3 节的算法 2DRANDOMBSP处理每块叶子区域内的线段片段。

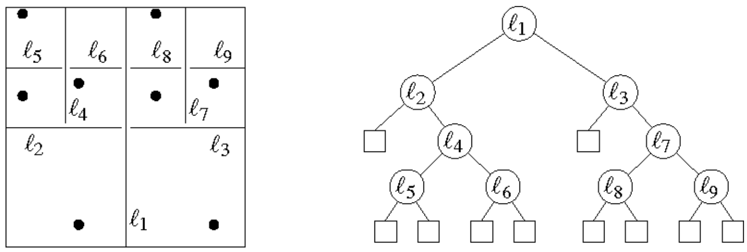


图12-23 由点集G(S)的四叉树划分可导出一棵BSP树

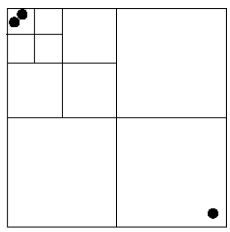


图12-24 若初始正方形U的某个角落附近有两个哨兵，而且它们相距极近，则划分出来的叶子区域可能会很多

上述算法的关键特性在于，对于低密度的场景，经前一阶段得到的叶子区域，都只与少量的物体相交。不幸的是，这里存在一个问题：叶子区域本身的数目可能很多。出现这个问题的情形之一就是，初始正方形 U 的某个角落附近有两个哨兵，而且它们相距极近。故此，需要进一步调整算法的前一阶段，确保其生成的叶子区域数量不超过线性。具体做法如下。

首先一项调整是：适当选取一个参数 k，任一区域一旦只包含不超过 k 个（而不是一个）哨兵，则不再细分。为何如此调整，以及具体如何选取 k 值，稍后再做讲解。

另一项调整如下。假设在递归划分的过程中，需要对某一正方形σ做细分。考察σ的四个象限，若其中的至少两个象限的内部含有超过k个哨兵，则沿用此前的方法实施一次四叉树分裂（quadtree split）：如图 12-25 所示，先用一条垂直线$l_v(\sigma)$将其一分为二，再用一条水平线$l_h(v)$将其二分为四。分裂之后，再递归地处理各个象限。如果每个象限所含哨兵均不超过k个，也要执行一次四叉树分裂——此时，四个象限都成为叶子区域。如果只有其中一个象限σ'内含多于k个哨兵，则需要小心处理——此时，有可能所有的哨兵都聚集在某个角落，从而需要经过很多次的分裂才能将它们分离开（参见〖引理 14.1〗）。为此，需要做一次紧缩（shrinking）操作。该操作的直观效果是，不断压缩σ'，直到至少有k个哨兵落到σ'的外部。更准确地，紧缩的过程如下。不妨以σ的左上象限σ'为例，其余三

种情况相仿。σ'的紧缩，也就是沿对角线方向（以保证σ'始终是正方形）、朝着左上角移动其左下角，直到至少有k个哨兵落在σ'之外。如果不那么严格的话，不妨仍用σ'指代紧缩之后的象限。可以看出，σ'的边界上至少穿过一个哨兵。接下来，如 图 12-25 所示，先沿着σ'右边所在的垂直线$l_v(\sigma)$将σ一分为二，然后沿着σ'底边所在的水平线$l_h(\sigma)$将它们二分为四。如此，σ被划分为四个区域，其中的两个为正方形。特别地，唯一可能包含多于k个哨兵，且因此需要进一步细分的区域σ'，必为正方形。

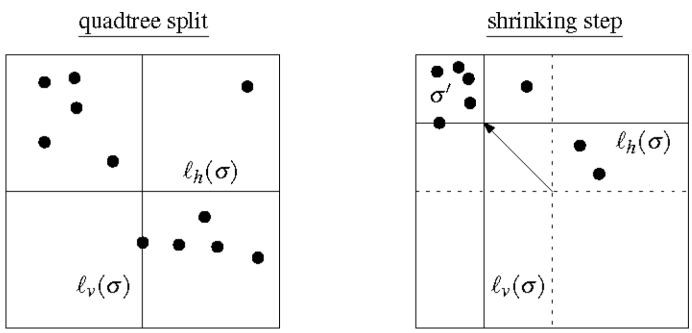


图12-25 四叉树分裂的实例，以及k = 4时的一次紧缩实例

以上描述，可以归纳为算法 PHASE1：

```
算法 PHASE1(σ, G, k)
输入：区域 σ，σ 内部的哨兵集 G，以及整数 k≥1
输出：BSP 树 T，其中每块叶子区域所含哨兵不超过 k 个
1.  if card(G) ≤ k
2.     then 创建仅含单个叶节点的一棵 BSP 树 T。
3.     else if σ 的四个象限中只有一个的内部包含多于 k 个哨兵
4.             then 按照以上介绍的紧缩操作，确定划分线 lv(σ)和 lh(σ)。
5.             else 按照以上介绍的四叉树分裂操作，确定划分线 lv(σ)和 lh(σ)。
6.          创建拥有三个内部节点的一棵 BSP 树 T:
               根节点以 lv(σ)为划分线；
               根节点的两个孩子以 lh(σ)为划分线。
7.          对 T 的每个叶节点μ及其对应区域中所含的哨兵
               递归构造一棵 BSP 树 Tμ；
               用 Tμ替换 T 中对应的叶节点。
8.  return T
```

〖引理 12.7〗

PHASE1(U, G(S), k)所构造的 BSP 树中，叶子数不超过 $O(n/k)$，其中每块叶子区域至多与 k+4λ个物体相交。

〖证明〗

首先证明叶子数的上界。这一数目比内部结点的数目多一，因此只需确定后者的上界。

若 card(G) = m，则将 PHASE1(σ, G, k)所构造 BSP 树中内部结点的最大数目记作 N(m)。若 m≤k，则无需划分，故此时有 N(m) = 0。否则，需要对 σ 实施四叉树分裂或紧缩操作，从而生成三个内部结点和四个子区域（这些子区域还需要进一步递归处理）。将这四个子区域所含哨兵的数目分别记作 $m_1, \ldots, m_4$，令 $I := \{i : 1 \le i \le 4 \text{ 且 } m_i > k\}$。包含 k 个或更少哨兵的区域必是叶子区域，因此对所有的 $i \in I$ 都有 $N(m_i) = 0$。于是：

$$N(m) \le \begin{cases} 0 & (\text{若} m \le k) \\ 3+\sum_{i\in I} N(m_i) & (\text{否则}) \end{cases}$$

以下将归纳证明 $N(m) \le \max(0, (6m/k)-3)$。对于 m≤k 这一点显然，故不妨假定 m>k。每个哨兵至多只能落在一个区域的内部，故有$\sum_{i\in I} m_i \le m$。若 σ 的四个象限中至少有两个包含多于 k 个哨兵，即 card(I) ≥ 2，则有

$$N(m) \le 3 + \sum_{i\in I} N(m_i) \le 3 + \sum_{i\in I} 6m_i/k - \text{card}(I)\cdot 3 \le 6m/k - 3$$

这与归纳命题吻合。若每个象限所含的哨兵均不超过 k 个，则对应的四块区域都是叶子区域，于是 N(m) = 3。考虑到 m>k 这一假设，可得 $N(m) \le (6m/k) - 3$。至此只剩一种情况——仅有一个象限包含多于 k 个哨兵。这种情况下需要实施紧缩操作。根据紧缩操作的做法，经紧缩的象限所包含的哨兵必少于 m-k 个，且其余的区域至多 k 个。于是，这种情况下有：

$$N(m) \le 3 + N(m-k) \le 3 + (6(m-k)/k - 3) \le 6m/k - 3$$

总而言之，正如归纳命题所言，无论何种情况都有 $N(m) \le 6m/k - 3$。至此，内部节点数目的上界得证。

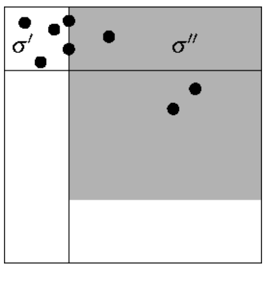


图12-26 每个非叶子区域σ''都用一个内含k个哨兵的正方形（阴影区域）来覆盖

下面再证明，每块叶子区域至多与k+4λ个物体相交。按照构造的算法，每块叶子区域的内部至多包含k个哨兵。故此，根据〖引理 12.6〗，每个正方形的叶子区域与k+4λ个物体相交。但是，鉴于并非所有叶子区域都是正方形，无法直接应用〖引理 12.6〗。

如图 12-25 所示，任一非正方形叶子区域σ''，都是某次紧缩操作的产物。根据约定，一旦有k或更多个哨兵不再落在被紧缩象限σ'的内部，紧缩操作旋即终止。这些哨兵中，至少有一个被σ'的边界穿过，故而实际上落在σ'外部的哨兵（即不计入那些恰好落在σ'边界上的那些哨兵）应不足k个。这就意味着，可以用一个内部包含k个哨兵的正方形来覆盖σ''（如图 12-26 中的阴影区域所示）。根据〖引理 12.6〗，该正方形至多与k+4λ个物体相交，而与σ''相交的物体也不会超过这个数目。 ❑

使用一个大于 1 的k值为何有利，〖引理 12.7〗做出了解释——k值越大，最终得到的叶子区域越少。当然，另一方面，k越大，每块叶子区域之内的物体也更多。因此，选取最佳的k值应遵循以下原则：既能尽可能地减少叶子区域，同时也不至于显著地增加各块叶子区域包含的物体。实际上，令k := λ即可——如此，叶子区域的数目（相对于与k = 1 时）将减少λ倍，同时就渐进意义而言，每块叶子区域内所含物体的最大数目也不致于增加（确切地说，不过是由 1+4λ增至 5λ）。

不过，还有一个问题：我们并不知道输入场景的密度λ，故而不能直接将其作为算法的参数。为此需要借助如下技巧。首先，猜测一个不大的λ，比如令λ = 2。以此λ作为参数 k 调用算法 PHASE1，并检查与所生成 BSP 树中每块叶子区域相交的物体是否超过 5k。若未超过，则转入算法的后一阶段；否则，将猜测的λ值加倍，重新尝试。如此可以归纳为以下算法：

算法 LOWDENSITYBSP2D(S)

输入： 平面上 n 个物体构成的集合 S

输出： 与 S 对应的一棵 BSP 树

```
1.  令 G(S)为 S 中所有 n 个物体共 n 个包围框的 4n 个顶点组成的集合。
2.  k ← 1；done ← false；U ← S 的包围框
3.  while not done
4.      do   k ← 2k；T ← PHASE1(U, G(S), k)；done ← true
5.           for T 中的每匹叶子μ
6.               do   找出μ所对应区域中的所有物体碎片，组成集合 S(μ)
7.                    if card(S(μ)) > 5k then done ← false
8.  for T 中的每匹叶子μ
9.      do   构造 S(μ)所对应的一棵 BSP 树 T_μ，并以 T_μ替换μ
10. return T
```

若输入的S由互不相交的线段组成，则以上的第 9 行可以调用 2DRANDOMBSP算法以构造BSP。如此，可以得到以下结论：

〖定理 12.8〗

由平面上任意 n 个互不相交物体组成的每一个集合 S，都拥有一个规模为 $O(n\log\lambda)$的 BSP，其中λ为 S 的密度。

〖证明〗

由〖引理 12.7〗，在PHASE1(U, G(S), k)所构造的BSP树中，每块叶子区域至多与k+4λ个物体相交。故此，只要k≥λ，则LOWDENSITYBSP2D算法第 7 行的逻辑测试就必然为false。（若k<λ，则该测试取true或false不定。）如此，不断增长的k一旦首次超过λ，while循环旋即终止。因为这里约定每次将k值加倍，所以在进入算法后一阶段（第 8 行）时，必有k≤2λ。

将执行到第 8 行时的k值记作k^*，由以上推断必有$k^*\le 2\lambda$。第 7 行的逻辑测试可以保证，每块叶子区域至多与 $5k^*$条线段相交。因此根据〖引理 12.1〗，只要第 9 行的确是调用 2DRANDOMBSP算法，则树T_μ的期望规模为$O(k^*\cdot\log k^*)$。考虑到叶子区域总共不过$O(n/k^*)$块，BSP树的总体规模将为$O(n\log k^*)$。既然$k^*\le 2\lambda$，本定理即得证。 ❑

〖定理 12.8〗所给出的上界，绝不会比$O(n\log n)$更差。也就是说，就最坏情况而言，上述算法与第 12.3 节所介绍的算法一样好；而当输入场景的密度很低时，新的算法将更为有效。

此前之所以要引入密度（density）的概念，是因为在最坏情况下，$\mathbb{R}^3$ 中一组三角形的BSP可能达到平方量级的规模。以上算法非常适用于平面线段集——最坏情况下，其构造的BSP规模为$O(n\log n)$；而输入场景为常数密度时，则为$O(n)$。那么，若用该算法来处理$\mathbb{R}^3$ 中的三角形集合，效果又将如何呢？事实正如以下定理（证明见 习题 12.18）所指出的，此时的计算效率同样很高。

〖定理 12.9〗

由 $\mathbb{R}^3$ 中任意 n 个互不相交三角形组成的每一个集合 S，都拥有一个规模为 $O(n\lambda)$的 BSP，其中λ为 S 的密度。

随着λ从 1 到n的不同取值，〖定理 12.9〗所给出的上界将从$O(n)$流畅地过渡到$O(n^2)$。因此首先，该算法所构造BSP的规模，是最坏情况最优的。但这一结果还为强大，实际上$O(n\lambda)$的上界对于λ的所有取值都是最优的——因为，对于满足 $1\le\lambda\le n$的任意n和λ，$\mathbb{R}^3$ 中都存在密度为λ的一组共n个三角形，与之对应的每一个BSP的规模都是$\Omega(n\lambda)$。

12.6 注释及评论

BSP树结构在许多应用领域都很流行，在计算机图形学中尤其如此。本章所提及的应用，是利用画家算法 [185]完成隐藏面消除。还有很多其它的应用，比如阴影生成 [124]、多面体的集合操作[292][370]，以及为实现交互式漫游而进行的可见性预处理 [369]。在运动规划的单元分解方法 [36]、区域查找（range searching）[60]都用到了这种结构，而GIS则将其作为一种通用的索引结构（index structure）。另外两种广为人知的结构——kd-树与四叉树——其实都不过是BSP树的特例，它们都限定划分平面必须是正交的。本书第 5 章和第 14 章分别详细介绍了kd-树和四叉树。

从理论角度对BSP树的研究，始自Paterson和Yao[317]；本章第 12.3 和 12.4 节介绍的结果，就来自于他们的论文。他们还证明了高维BSP的上界——$\mathbb{R}^d$中任何一组(d-1)-维单纯形（simplex）,$d \geq 3$，都存在一个规模不超过$O(n^{d-1})$的BSP。Paterson和Yao[318]还针对高维空间中的正交物体（orthogonal object）得出了一些结论。例如，他们证明：对于$\mathbb{R}^3$中的任何一组正交矩形，都存在一个规模不超过$O(n\sqrt{n})$的BSP；而且，在最坏情况下这个上界是紧的。以下，将介绍其后得出的若干新成果。更为详细的介绍，可以参考Toth[373]。

曾经许久悬而未决的一个问题是：平面上任意n条互不相交的线段，是否拥有一个规模为$O(n)$的BSP？Toth[372]构造了一组线段，并证明它的每个BSP的规模都不会低于$\Omega(n\log n/\log\log n)$，从而否定了上述猜测。请注意，相对于目前已知的上界$O(n\log n)$，这一下届仍有距离。在几种特殊情况下，都可以保证BSP的规模不超过$O(n)$。例如，Paterson和Yao[317]证明：平面上任意一组互不相交的正交线段，必然拥有一个$O(n)$规模的BSP。d'Amore和Franciosa[138]也得出了相同的结论。Toth[371]则将这一结果推广至方向仅限于几种可能的线段集。拥有线性规模BSP的其它物体集特例还包括：长度接近的一组线段 [54]，以及正如本章已证明的，具有常数密度的一组物体。

第 12.5 节之所以分析了低密度的场景，是受到以下现象的启发：在三维空间中，BSP在最坏情况时的规模不能反映其实际的运行效率。在对几何算法做分析时，时常会发生类似的问题：尽管我们可以找到使算法运行低效的输入实例，但现实中遇到此类输入的可能性不大。于是引发两方面的问题：首先，在判断该算法是否实用时，依最坏情况所做的分析结论没有太多参考价值；另外，因为在设计算法时通常都着眼于保证能够最佳地应对最坏情况，所以往往会把算法搞得异常复杂，以便应对实践中根本就不会出现的情况。这一问题背后的原因在于：集合算法的执行时间，通常不仅仅取决于输入的规模，而是与输入物体的外形及其在空间中的分布密切相关。解决此类问题的方法之一，就是定义一个能够刻画输入数据几何特征的参数——正如第 12.5 节那样。

丰满度（fatness）是在此类问题中十分常用的一个参数。任一物体如果角度不小于β，就被称作是β丰满的（β-fat）。[268]已经证明：若β为常数，则对于平面上任意n个β-丰满的相交三角形，其并

集的复杂度与n成近似线性关系。截止目前，最好的上界为$O((1/\beta)\cdot\log(1/\beta)\cdot n\cdot\mathbf{loglog}n)$[314]。丰满度的概念，已被推广至任意凸的物体，甚至非凸的物体。丰满度最具概括性的定义是由van der Stappen[362]给出的——$\mathbb{R}^d$中所谓一个β-丰满的物体o，必须具有如下性质：对于中心落在o内部但又不是完全包含于o内部的任一球B，总有$vol(o\cap\beta) \geq \beta\cdot vol(B)$，其中vol(·)表示体积。在很多问题来中，处理丰满的物体，要比处理一般性物体更为高效。这种问题包括区域查找（range search）与点定位（point location）[51][60]、运动规划（motion planning）[363]、隐藏面消除（hidden-surface removal）[229]、光线发射（ray shooting）[21][49][53][228]以及深度序（depth order）的计算 [53][228]。

关于第 12.5 节所使用的密度（density）参数本身，也有很多的研究。[55][362]已证明：任意一组互不相交的β-丰满的物体，密度不超过$O(1/\beta)$。因此，与针对低密度场景的任何结论相对应地，都可以直接得到一个针对互不相交的丰满物体的结论。第 12.5 节针对低密度场景而介绍的BSP构造算法，是de Berg[51]构造算法的修改和改进版本。以上所提及的针对丰满物体的一些结论，实际上正是基于该构造算法 [60][363]，故而也可以应用于低密度场景。

12.7 习题

习题 12.1 试证明：PAINTERSALGORITHM 是正确的。即需要证明：如果物体 A（的某部分）先于物体 B（的某部分）被扫描转换，那么 A 就不可能挡在 B 的前面。

习题 12.2 设 S 为平面上一组共 m 个多边形，其中共有 v 个顶点。设 $\mathcal{T}$ 为 S 的一棵规模为 k 的 BSP 树。试证明：该 BSP 所生成碎片的总体复杂度为 $O(n + k)$。

习题 12.3 试给出平面上一组线段的例子来说明：若采用贪婪算法（greedy algorithm，即每次都选用导致最少切分的直线 l(s)进行分割）来构造自动划分，则最终所得 BSP 的规模会达到平方量级。

习题 12.4 试举例说明：对于平面上包含 n 条互不相交线段的某个集合 S，虽然的确存在一棵规模为 n 的 BSP 树，但是 S 的任何一个自动划分的规模却至少是$\lfloor 4n/3 \rfloor$。

习题 12.5 试举例说明：平面上包含 n 条互不相交线段的某个集合 S，其任何自动划分的深度都至少是$\Omega(n)$。

习题 12.6 我们已经证明过：就 2DRANDOMBSP 算法所生成的划分而言，其期望规模为 $O(n\mathbf{log}n)$。那么，其在最坏情况下的规模呢？

习题 12.7 假设将 2DRANDOMBSP 算法应用于平面上一组允许相交的线段。关于由此生成的 BSP 树的规模，你可以有什么结论？

习题 12.8 试考虑算法 3DRANDOMBSP2。在引入一张分割平面的时候，如何才能找到那些必须被切分的单元？进而，如何才能有效地实施分割？对于这两个问题，我们并没有做出回

答。试对这一步的详细实现过程做一描述，并对你所提出的算法的运行时间做一分析。

习题 12.9 试给出一个分治式确定性算法，为平面上任意 n 条线段，构造规模为 $O(n\log n)$的 BSP 树。提示：尽可能多地利用免费分割；只有在免费分割不可行时才使用垂直分割线。

习题 12.10 设 C 为平面上一组共 n 个互不相交的单位圆盘（即半径为 1 的圆盘）。试证明：存在一个与 C 对应的 BSP，其规模为 $O(n)$。提示：在一开始时，适当的选用一组垂线——这些直线的形式都是 x = 2i，其中 i 为某个整数。

习题 12.11 借助 BSP 树，可以完成各种各样的任务。任意给定一个平面子区域划分，假设对应于其中各边，我们已经构造出了一个 BSP。

a. 试给出一个算法，利用 BSP 树在该子区域划分中进行点定位。在最坏情况下，查询时间是多少?

b. 试给出一个算法，对于任一待查询线段（query segment），利用 BSP 树从该子区域划分中报告出与该线段相交的所有面。在最坏情况下，查询时间是多少?

c. 试给出一个算法，对于任一与坐标轴平行的待查询矩形，利用 BSP 树从该子区域划分中报告出与该矩形相交的所有面。在最坏情况下，查询时间是多少?

习题 12.12 第 5 章已经对kd-树做过介绍。kd-树不仅能够用来存放点，也能存储线段。在这种情况下，它实际上就是一种特殊的BSP树——其中，对于树中处于偶数层的每个节点，分割线都是水平的；而奇数层则都是垂直的。

a. 与kd-树相比，BSP树具有哪些优点及缺点？试就此做一讨论。

b. 在平面上，对于任意两条互不相交的线段，都存在一棵规模为 2 的BSP树。试证明：不可能存在任何常数c，使得对于平面上任意两条互不相交的线段，都存在一棵规模不超过c的kd-树。

习题 12.13 试证明：平面上任意一组互不相交的圆盘，密度不会超过 9。（故此，依赖关系将与圆盘的数目无关。）进而由此证明：对于平面上任意一组共 n 个互不相交的圆盘，其 BSP 的规模为 $O(n)$。将此结论推广至高维空间。

习题 12.14 所谓α-丰满（α-fat）的三角形，就是要求它的三个角都不小于α。试证明：平面上任意一组互不相交α-丰满的三角形，密度不超过 $O(1/\alpha)$。进而由此证明：平面上任意一组互不相交的α-丰满三角形，都有一个规模不超过 $O(n\log(1/\alpha)$的 BSP；这将意味着，任意一组共 n 个互不相交的正方形，都有一个规模不超过 $O(n)$的 BSP。

习题 12.15 试构造密度为常数的一组共n个三角形，要求其任何一个自动划分（auto-partition）——亦即所采用的划分面必须与输入三角形重合的划分——的规模至少为$\Omega(n^2)$。（这个实例将说明，即使输入三角形集的密度为常数，第 12.4 节所介绍的随机算法依然可能构造出期望规模为$\Omega(n^2)$的BSP。）提示：考虑均与z-轴平行、到xy-平面的投影构成网格的一组三角形。

习题 12.16 设T为平面上互不相交的一组三角形。若不选用所有三角形的包围框（bounding box）

的顶点作为哨兵，还可以选用三角形本身的顶点。试针对〖引理 12.6〗举一反例，说明这种选取哨兵的做法并不好。你是否可以举出这样的反例，同时满足所有三角形都是等边三角形（equilateral triangle）？

习题 12.17 试证明：只要实现得合理，算法 LowDensityBSP2D 的运行时间可以控制在 $O(n^2)$ 以内。

习题 12.18 试将算法 LowDensityBSP2D 推广至 $\mathbb{R}^3$，并就对应的 BSP 的规模做一分析。

13

机器人运动规划：随意所之

机器人学（robotics）的最终目标之一，就是设计出独立自主的机器人（autonomous robot）——在指挥这种机器人时，我们只需告诉它去做*什么*，而不必告诉它*如何*去做。其中尤为重要的一个方面就是，机器人应该懂得如何为自己的运动进行规划。

运动规划前，机器人必须了解其所处的环境。例如，工厂中的移动机器人必须事先知道障碍物的分布。其中诸如墙或机床的位置等信息，可以描述为一张平面建筑图（floor plan）。其它的信息，则需要机器人借助传感器获得。遇到建筑图上未予标定的障碍物（比如说人）时，机器人也应能够

发现。根据对所处环境的了解，机器人应能够顺利抵达目的地，而不致在沿途发生任何碰撞。

无论哪种机器人，只要它希望在真实的世界中移动，就必须求解运动规划问题。 以上介绍的情况，假设是一个独立自主的机器人，它需要在某个工厂的环境中来回运动。这种机器人依然十分罕见，更为常见的是机械手（robot arm），在今天的工业环境中，它们已经得到了广泛的应用。

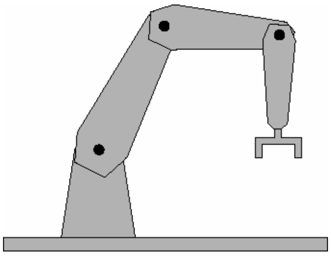


图13-1 机械手

如 图 13-1 所示，所谓的机械手——或称作多关节型机器人（articulated robot）——由若干段杆件（link）通过关节（joint）联接而成。通常，机械手的一端固定在工作平面上——称为底座（base）；另一端则装有手柄（hand）或者某种工具。杆件的数目从三至六段不等，有的甚至更多。关节通常不外乎两种：旋转式关节（revolute joint）或柱状关节（prismatic joint）。前者允许杆件围绕关节任意转动；而后者只允许一段杆件（相对另一段杆件）滑进、滑出。机械手大多被用以组装或者操纵某些零件，或用来完成焊接或喷漆等任务。为此，它们应能够往返移动于不同的位置之间，而不致与周围环境或正在操作的物体发生碰撞。还有一点既有趣也复杂——它们也不应与自己发生碰撞。

本章中将介绍运动规划中的一些基本概念及技术。鉴于一般性的运动规划问题十分难解，故需要做一些简化假设。

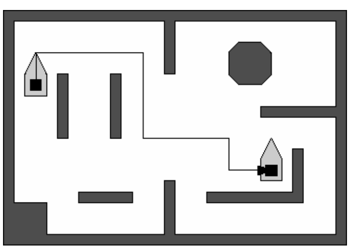


图13-2 限制于平面上特定区域内运动的机器人

最大的一点简化是，这里只讨论二维的运动规划问题。也就是说，如 图 13-2 所示，运动的环境是平面上的一个区域；而且，其中障碍物的外形都是多边形；另外，机器人本身的外形也是多边形。这里还假定：环境是静态的——亦即，在机器人运动的沿途不会遇到行人；而且，机器人对环境的情况事先已知。将机器人限制为平面形式，看似很苛刻，实则不然——对于一个在厂房中运动的机器人来说，只要平面建筑图能够提供墙壁、机床等信息，常常就已经足以进行运动规划了。

机器人可执行哪些运动，取决于其机械结构。有的机器人可沿任何方向移动，有的则只能做受限的运动。例如，汽车式（car-like）机器人就不能横行——否则，将多辆汽车并排停放就轻而易举了。此外，机器人的转弯半径（turning radius）通常都不能太小。汽车式机器人运动的几何关系十分复杂，因此我们把讨论的范围仅限制于可沿任一方向移动的机器人。实际上，这里主要研究的是那些只能进行平移的机器人；本章的最后，将简要地讨论那种可以通过旋转改变自身方向的机器人。

13.1 工作空间与 C-空间

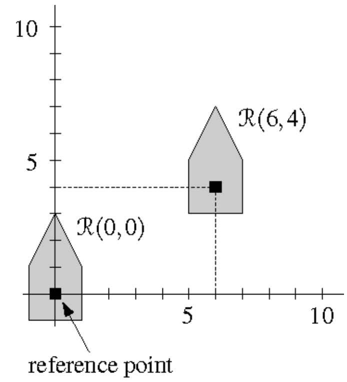


图13-3 用平移向量表示（平移）机器人的位置（以左下角原点为参考点）

设$\mathcal{R}$为在二维环境中移动的一个机器人。机器人所处的环境也称为工作空间（work space），它由一组障碍物S = {P_1, …, P_t}组成。我们假定，$\mathcal{R}$本身是一个简单多边形（simple polygon）。这样，如 图 13-3 所示，机器人所处的每个位置（placement或configuration）都可以表示为一个平移向量。如果机器人沿向量(x, y)做了一次平移，就记之为$\mathcal{R}$(x, y)。例如，设某个机器人所对应多边形的顶点分别为(1, -1)、(1, 1)、(0, 3)、(-1, 1)和(-1, -1)，则$\mathcal{R}$(6, 4)所对应的顶点就是(7, 3)、(7, 5)、(6, 7)、(5, 5)和(5, 3)。采用这种记号，可以用$\mathcal{R}$(0, 0)的所有顶点来表示一个机器人。

也可以按照所谓参考点（reference point）的概念来理解这一点。当 $\mathcal{R}$(0, 0)的内部包含原点(0, 0)时，这是一种最直观的理解方式。根据定义，这个点被称为该机器人的参考点。无论机器人在任何给定的位置，只要给出参考点的坐标，就可以定义出机器人此时的位置。这样，$\mathcal{R}$(x, y)的含义就是"按照机器人当前的位置，其参考点位于(x, y)"。一般而言，参考点并不一定落在机器人的内部；实际上，它可以落在机器人之外——这种情况可以想象为，用一根"隐身的"棍棒将机器人与它的参考点联接起来。根据定义，$\mathcal{R}$(0, 0)的参考点位于坐标原点。

现在，假设机器人能够通过旋转（比如说绕着它的参考点旋转）改变方向。此时，如 图 13-4 所示，需要一个新的参数ϕ来描述机器人的方向。如果机器人的参考点在(x, y)，而且已经沿逆时针方向旋转了ϕ角，就可以用$\mathcal{R}$(x, y, ϕ)来描述它。因此，最初确定的是$\mathcal{R}$(0, 0, 0)。

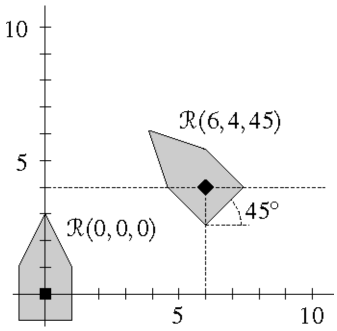


图13-4 若机器人可旋转，则需要引入角度参数以描述其所处位置

一般而言，描述机器人位置的一组参数，分别对应于机器人的几个自由度（degree of freedom – DOF）。对于只能在平面上平移运动的机器人来说，自由度为二；如果机器人既能平移也能旋转，其自由度就是三。当然，如果是在三维空间中，就需要更多的参数来描述机器人：$\mathbb{R}^3$ 中只能平移的机器人自由度为三，$\mathbb{R}^3$ 中既能平移也能旋转的机器人自由度为六。

机器人 $\mathcal{R}$ 的参数空间，通常被称为 C-空间（configuration space），记作 $\mathcal{C}(\mathcal{R})$。C-空间中的每个点 p，分别对应于机器人在工作空间中的某一位置 $\mathcal{R}(p)$。对于在平面上可平移、可旋转的机器人来说，C-空间是三维的。在这个空间中，任何一点(x, y, ϕ)都对应于工作空间中的某个位置 $\mathcal{R}(x, y, \phi)$。C-空间并不是一个三维欧氏空间；这个空间仅仅是 $\mathbb{R}^2 \times [0 : 360)$。因为旋转 0 度与旋转 360 度是等价的，所以机器人的 C-空间具有一种特殊的拓扑，这种拓扑与圆柱面类似。

对于可在平面上做平移运动的机器人，其 C-空间的确是一个二维欧氏空间，因此该空间与工作空间是一样的。尽管如此，还是应该区分这两个概念：所谓的工作空间，是机器人在其中实际运动的空间——亦即真实的世界；而所谓 C-空间，是机器人的参数空间。工作空间中的每个多边形机器人，都可表示为 C-空间中的一点；C-空间中的每个点，都对应于工作空间中某个实际机器人的位置。

如此，便给出了一种定义机器人位置的方法——给出用来确定（机器人）位置的参数值；换而言之，也就是在 C-空间中给出一个点。不过显然，在 C-空间中，并非每个点所指示的位置都是可能的——如果处于某个点所指示位置的机器人会与 S 中的障碍物相交，则这个点就应该被禁止。C-空间中由这类点所构成的部分，被称为禁止 C-空间（forbidden configuration space），或者简称禁止空间（forbidden space），记作 $\mathcal{C}_{forb}(\mathcal{R}, S)$。在 C-空间中其余的部分，每个点都对应于某一自由位置（free placement）——也就是说，处于该位置的机器人不会与任何障碍物相交。这些部分合起来被称为自由 C-空间（free configuration space），或者简称自由空间（free space），记作 $\mathcal{C}_{free}(\mathcal{R}, S)$。

机器人的每条运动路径，都可以被映射为C-空间中一条曲线。反之亦然——路径上的每一点都可以相应地映射为C-空间中的某一点。每条无碰撞的路径都可以映射为自由空间中的某条曲线。如图 13-5 所示的，就是一个沿平面做平移运动的机器人。图中左侧显示的是工作空间，在机器人的起

点与终点之间，有一条无碰撞的路径。图中右侧显示的是C-空间，其中灰色的区域就是对应的禁止空间，介于灰色区域之间的部分就是自由空间。在C-空间中，各障碍物已经没有什么意义，不过为了说明清楚，还是将它们画上去了。其中也画出了与这条无碰撞路径相对应的一条曲线。

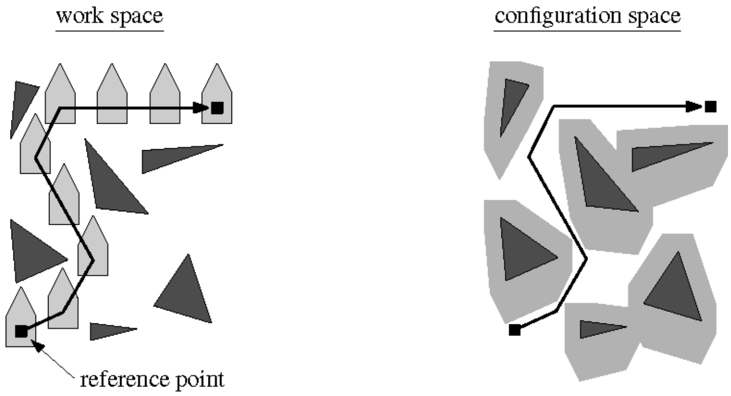


图13-5 工作空间中的每条路径，对应于C-空间中的某条曲线：工作空间（左），C-空间（右）

上面介绍了如何将机器人的任一位置映射为 C-空间中的一点，以及如何将机器人的任一运动路径映射为 C-空间中一条曲线。那么，是否也可以将障碍物映射到 C-空间中去呢？可以。任一障碍物 P 都可以映射为 C-空间中的某一点集，该点集由所有满足“R(p)与 P 相交”的点 p 组成。这个集合称作与 P 对应的 C-空间障碍物（configuration space obstacle），或简称为 C-障碍物（C-obstacle）。

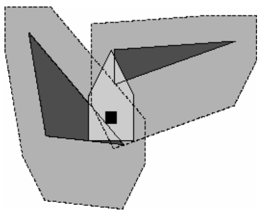


图13-6 一对C-障碍物相交，当且仅当存在某个位置，处于该位置的机器人与相应的一对障碍物相交

即使在工作空间中的障碍物没有相交，它们在C-空间中对应的C-障碍物也可能会相交。如 图 13-6 所示，如果机器人在某一位置同时与至少两个障碍物相交，就会发生这种情况。

至此，我们一直都忽略了一处细节：如果机器人恰好与某个障碍物相切，这种情况也应算作发生碰撞吗？换而言之，从拓扑的角度来看，究竟应该将障碍物定义为开集，还是闭集？在后续的讨论中，我们将采用前一种方案：所有障碍物都是开集——按照这种定义，机器人与障碍物相切是允许的。就本章的内容来说，这一点无关紧要，但是在第 15 章中，这将是很关键的一点。在实际应用

中，如果在某次运动的过程中，机器人需要与某个障碍物在短距离内贴身而过，都将被视为不安全的——因为对机器人的控制可能会存在误差。这类运动是可以避免的，比如我们可以在进行路径规划之前，将每个障碍物稍微放大一点。

13.2 点机器人

在着手平面多边形机器人的运动规划之前，首先讨论点机器人的情况。根据前一节所介绍的从工作空间到 C-空间的映射，这是很自然的想法。此外，从简单情况入手总是个好方法。按照此前的习惯，我们将机器人记作 $\mathcal{R}$，将各障碍物记作 $\mathcal{P}_1, \ldots, \mathcal{P}_t$。所有障碍物的形状都是多边形，而且它们的内部互不相交；各多边形所含顶点的总数记作 n。对于点机器人来说，工作空间与 C-空间是完全相同的。（之所以这么说，是因为只要自然地将参考点当作点机器人本身，这两个空间就会完全一致；即使是采用其它参考点，只要经过一次平移，C-空间就会与工作空间完全吻合。）

我们并不是直接去构造从给定起点通往给定终点的一条路径，而是首先建立起某种数据结构，来存储有关自由空间的描述信息。此后，就可以利用这一数据结构，在任意给定的起点和终点之间构造出一条路径。如果机器人所处的工作空间是固定的，而且需要反复多次规划路径，那么这种方法将很有用处。

为了简化叙述，我们将机器人的运动范围限制在一个包围框 B 中。这个包围框应该足够大，以容纳所有的多边形。换而言之，需要引入一个附加的无穷大障碍物——即 B 之外的整个范围。这样，自由 C-空间 $\mathcal{C}_{\text{free}}$ 就是 B 中未被任何障碍物覆盖的那些区域：

$$\mathcal{C}_{\text{free}} = B \setminus \bigcup_{i=1}^{t} \mathcal{P}_i$$

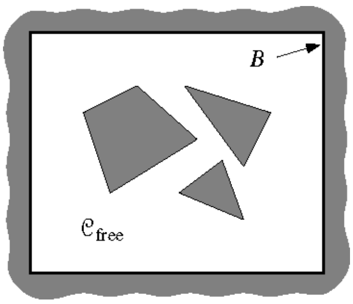


图13-7 自由空间

如 图 13-7 所示，自由空间有可能包含多块互不连通的区域，其中也可能含有孔洞。我们的目标是构造出对自由空间的某种表示，以便此后在任意的起点和终点之间规划路径。为此，如 图 13-8 所示，我们采用了梯形图（trapezoidal map）的结构。根据第 6 章的介绍，对于散布于某个包围框中

的一组互不相交的线段，为了构造对应的梯形图，只要从每个线段端点各引出两条垂直射线：一条垂直向上，直到撞上某条线段（或者包围框）；另一条垂直向下，直到撞上某条线段（或者包围框）。

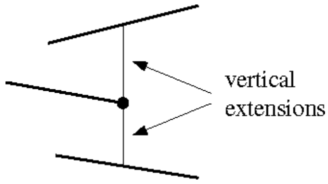


图13-8 将自由空间表示为梯形图

第 6 章曾介绍过一个随机算法TRAPEZOIDALMAP，可在O(**n**log**n**)的期望运行时间内，构造出n条线段的梯形图。下述算法的功能，就是以上述算法当作一个子程序，构造出自由空间的一个表示。

算法 COMPUTEFREESPACE(S)
输入：一组互不相交的多边形 S
输出：对于一个点机器人 $\mathcal{R}$，$\mathcal{C}_{free}(\mathcal{R}, S)$所对应的梯形图，
1. 令 E 为 S 中各多边形的所有边
2. 利用第 6 章所介绍的算法TRAPEZOIDALMAP，构造出梯形图$\mathcal{T}$(E)
3. 从$\mathcal{T}$(E)中将落在各多边形内部的那些梯形删除掉，返回所得到的子区域划分

这个算法如 图 13-9 所示。图中(a)部分所显示的，是包围框中所有障碍物边所对应的梯形图；它是由算法的第 2 行构造出来的。图中(b)部分所显示的，是第 3 行将各障碍物内部的梯形（trapezoid）删除之后所得到的子区域划分（subdividsion）。

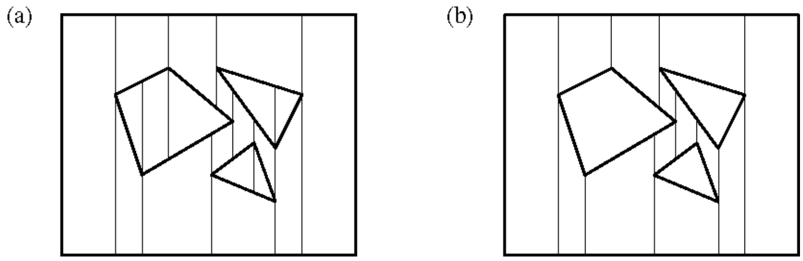


图13-9 构造自由空间的梯形图

有一点细节尚未做出说明：在将落在各障碍物内部的梯形删除之前，应该如何将它们找出来呢？这件事并不困难——在执行完 TRAPEZOIDALMAP 之后，实际上我们已经知道了每个梯形顶部的边界是那一条边，因此也就知道了这条边界属于哪一个障碍物。接下来，只要看看这条边究竟是属于该障碍物的上边界还是下边界，就可以判断出是否应该删除这个梯形。每一次这样的测试只需常数时间——因为每个障碍物的边都是沿着其边界有序存放的，所以障碍物相对每一条（有向）边是处于左侧还是右侧，都是确定而且已知的。

由于 TRAPEZOIDALMAP 的期望运行时间为 O(**n**log**n**)，我们可以得出如下结论：

〖引理 13.1〗

对于一个点机器人，若它运动的环境中包含一组互不相交的多边形障碍物，且障碍物总共包含 n 条边，则可以借助随机算法，在 $O(n\log n)$期望运行时间内构造出一幅描述其自由 C-空间的梯形图。

在后面的讨论中，我们把描述自由空间的梯形图记作 $\mathcal{T}(\mathcal{C}_{free})$。

借助 $\mathcal{T}(\mathcal{C}_{free})$，如何才能在起点 p_{start} 和终点 p_{goal} 之间规划出一条路径呢？

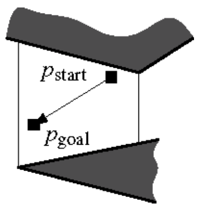


图13-10 起点、终点属于同一梯形的情况

如 图 13-10 所示，若p_{start}和p_{goal}都属于图中的同一梯形，则再容易不过了——机器人可以沿着一条线段，径直通往终点。

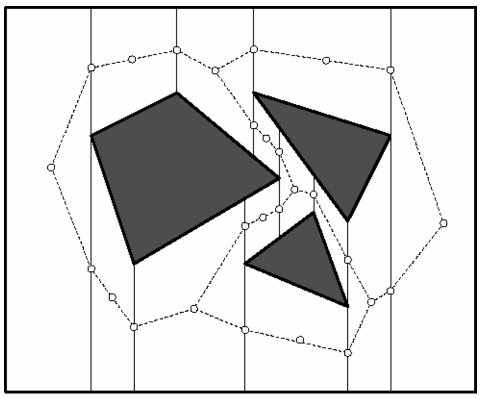


图13-11 路线图

然而，若起点和终点分属不同梯形，问题就不那么简单了。此时的路径会穿越多个梯形，而且在途经某些梯形时还可能拐弯。为引导这种跨越多个梯形的运动，需在自由空间中构造出一幅路线图（road map）。路线图实际上是嵌入于平面上（更准确地，是嵌入于自由空间中）的一幅图G_{road}。除首、尾两段之外，路径的其它部分都与路线图吻合。请注意，水平邻接的任何一对梯形都由一条垂直边分隔开来，该垂边必然是某一线段端点的延长线。这就启发我们按照以下方式定义路线图。在每个梯形的中心、每条垂直边的中点各放置一个节点。两个节点之间有一条弧相联，当且仅当其中一个节点处于某个梯形的中心，而另一个处于该梯形的边界上。在将这幅图嵌入于平面时，所有的弧都实现为直线段——亦即，路线图中的弧各对应于机器人的一段直线运动。如 图 13-11 所示。

只要对描述 $\mathcal{T}(\mathcal{C}_{free})$的双向链接边表（doubly-connected edge list）进行遍历（traversal），就可以在 $O(n)$时间内构造出路线图 $\mathcal{G}_{road}$。沿着路线图的各条弧，可以从一个梯形的中心，经过其边界上的一个节点，到达与其共同拥有这段边界的另一个梯形的中心。

将路线图与梯形图结合起来，即可在任意起点和终点之间进行运动规划。为此，首先要分别找出这两个点所属的梯形Δ_{start}和Δ_{goal}。若它们是同一个梯形，就可以从p_{start}沿着直线径直走向p_{goal}。否则，令v_{start}和v_{goal}分别为这两个梯形在$\mathcal{G}_{road}$中各自对应的节点。在p_{start}和p_{goal}之间待构造的那条路径，由三段组成：首先是从p_{start}到v_{start}的一段直线运动，然后是由路线图中若干条弧首尾衔接、联接于v_{start}与v_{goal}之间的一段折线运动，最后又是从v_{goal}到p_{goal}的一段直线运动。如 图 13-12 所示。

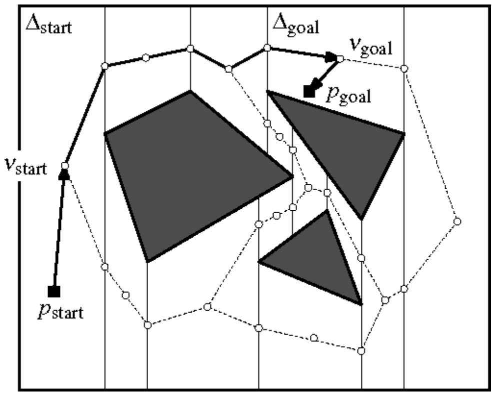


图13-12　根据图13-11中的路线图规划出来的一条路径

规划路径的具体过程，可以总结为如下算法：

> 算法 COMPUTEPATH($\mathcal{T}(\mathcal{C}_{free})$, $\mathcal{G}_{road}$, p_{start}, p_{goal})
>
> 输入：自由空间对应的梯形图 $\mathcal{T}(\mathcal{C}_{free})$，路线图 $\mathcal{G}_{road}$，以及起点 p_{start} 和终点 p_{goal}
>
> 输出：从 p_{start} 通往 p_{goal} 的一条路径（如果存在的话）
> 　　如果不存在这样的路径，也要报告这一情况
>
> 1. 分别找出 p_{start} 和 p_{goal} 所属的梯形Δ_{start}和Δ_{goal}
> 2. **if** (Δ_{start} 或Δ_{goal} 不存在)
> 3. 　　**then** 报告“起点或终点落在禁止空间中”
> 4. 　　**else** 令 v_{start} 为 $\mathcal{G}_{road}$ 中放置于Δ_{start} 中心处的节点
> 5. 　　　　令 v_{goal} 为 $\mathcal{G}_{road}$ 中放置于Δ_{goal} 中心处的节点
> 6. 　　　　利用广度优先搜索，
> 　　　　　在 $\mathcal{G}_{road}$ 中构造出一条从 v_{start} 通往 v_{goal} 的路径
> 7. 　　　　**if** (不存在通路)
> 8. 　　　　　**then** 报告“p_{start} 与 p_{goal}之间没有通路”

9.	**else** 报告出一条通路 (* 这条通路由三段组成： *) (* 1. 从 p_{start} 通往 v_{start} 的一段直线运动 *) (* 2. 在 $\mathcal{G}_{road}$ 中规划出来的一条路径 *) (* 3. 从 v_{goal} 通往 p_{goal} 的一段直线运动 *)

在对该算法的复杂度进行分析之前，我们先来检查其正确性。如此规划出来的路径，必然是无碰撞的吗？另外，只要这种无碰撞的路径的确存在，上面的算法就一定能够找出一条吗？

前一个问题很好回答：规划出来的路径必然是无碰撞的——因为，组成路径的每一条线段都落在某个梯形之内，而每个梯形都是自由空间的一部分。

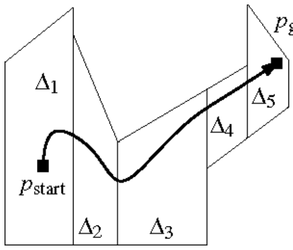


图13-13 只要存在无碰撞的路径，算法COMPUTEPATH就一定能够找出一条

为回答后一个问题，如 图 13-13 所示，假定在p_{start}和p_{goal}之间的无碰撞通路的确存在。于是，显然p_{start}和p_{goal}都必定分别落在自由空间中的某个梯形内，因此只需证明：在$\mathcal{G}_{road}$中存在一条从v_{start}通往v_{goal}的路径。沿着p_{start}与p_{goal}之间的通路，必然要穿越一系列的梯形。不妨将这些梯形记作$\Delta_1, \Delta_2, \ldots, \Delta_k$。根据定义，有$\Delta_1 = \Delta_{start}$和$\Delta_k = \Delta_{goal}$。令$v_i$为$\mathcal{G}_{road}$中位于$\Delta_i$中心处的节点。若$\Delta_i\Delta_{i+1}$ 属于路径上的一段，则Δ_i和Δ_{i+1} 必然是拥有一段公共垂直边界的一对邻居。然而根据$\mathcal{G}_{road}$的构造规则，这样的一对梯形必然会通过其共同边界上某一节点相联接。因此，在$\mathcal{G}_{road}$中存在一条（由两段弧联接而成的）从v_i到v_{i+1} 的通路。亦即，在v_1 和v_k之间也必然存在一条通路。于是，只要对$\mathcal{G}_{road}$进行广度优先搜索，就总是能够在v_{start}和v_{goal}之间找到某条通路（当然，具体是哪一条，可能不同）。

现在来分析上述算法的运行时间。

采用第 6 章所介绍的点定位结构，可以在$O(\log n)$时间内确定起点和终点所在的梯形。也可以直接地逐一检查所有的梯形，这样需要线性的时间——后面我们将看到，算法的其余部分必然需要线性的时间，因此这种方法并不会影响该算法的渐进复杂度。

广度优先搜索所需的时间，线性正比于图 $\mathcal{G}_{road}$ 的规模。而在这幅图中，对应于每个梯形、每条垂直延长线分别设有一个节点。实际上，无论是垂直延长线的数目还是梯形的数目，都线性正比于各障碍物所含顶点的总数。另外，既然这是一幅平面图，其中所含弧的条数也必然是线性（正比于

其中节点数目）的。因此，广度优先搜索需要 $O(n)$时间。

为了报告规划出来的路径而消耗的时间，不会超过 $\mathcal{G}_{road}$ 中该路径上弧的条数——即 $O(n)$。

这样，就得到了如下定理：

〖定理 13.2〗

设点机器人 $\mathcal{R}$ 运动于一组多边形障碍物 S 之间，各障碍物所含边的总数为 n。可以在 $O(n\mathbf{log}n)$期望运行时间内对 S 进行预处理，使得我们总可以在 $O(n)$时间内，在任何起点与终点之间为 $\mathcal{R}$ 规划出一条无碰撞的路径（如果的确存在这样一条路径的话）。

尽管本节介绍的算法所规划出来的路径必然是无碰撞的，但我们却不能保证得到的路径不会舍近求远地兜大圈子。第 15 章里将设计一个算法，真正找出可能的最短路径。不过，那个算法的速度将会慢一个数量级。

13.3 Minkowski 和

上一节解决了点机器人的运动规划问题：首先构造出其自由空间的梯形图，然后利用这张图来进行运动规划。对于多边形机器人，该方法依然适用。多边形机器人更难处理，因为它们与点机器人有一个重要的区别：C-空间中的障碍物已不再与工作空间中的障碍物一样。因此，有必要首先对平移式多边形机器人的自由 C-空间做一分析。下一节，再介绍自由 C-空间的构造方法，以及如何利用它为机器人进行运动规划。

假定机器人 $\mathcal{R}$ 是凸多边形（convex polygon），而且暂且假定所有障碍物都是凸的。你应该记得，如果机器人的参考点在(x, y)，就将 $\mathcal{R}$ 的位置记作 $\mathcal{R}(x, y)$。对于机器人 $\mathcal{R}$ 而言，每一障碍物 $\mathcal{P}$ 所对应的 C-空间障碍物（或简称为 C-障碍物），都被定义为 C-空间中的一个点集，如果 $\mathcal{R}$ 所处的位置对应于该点集中的某个点，$\mathcal{R}$ 就会与 $\mathcal{P}$ 相交。因此如果将 $\mathcal{P}$ 的 C-障碍物记作 $\mathcal{CP}$，就有

$$\mathcal{CP} := \{(x, y) \mid \mathcal{R}(x, y) \cap \mathcal{P} \neq \emptyset\}$$

为了画出$\mathcal{CP}$的形状，如 图 13-14 所示，可以让$\mathcal{R}$沿着$\mathcal{P}$的边界滑行一圈——$\mathcal{R}$的参考点所经过的轨迹曲线，就是$\mathcal{CP}$的边界。

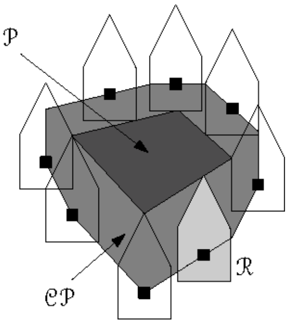


图13-14 若将$\mathcal{R}$沿$\mathcal{P}$的边界滑行一周，$\mathcal{R}$参考点的轨迹就是$\mathcal{CP}$的边界

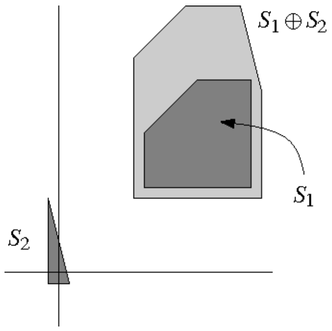


图13-15 点集的Minkowski和

也可通过别的方法描述这一过程，比如采用Minkowski和（Minkowski sum）的概念。如 图 13-15 所示，对于任何两个集合$S_1 \subset \mathcal{R}^2$和$S_2 \subset \mathcal{R}^2$，其Minkowski和（记作$S_1 \oplus S_2$）定义为：

$$S_1 \oplus S_2 \quad := \quad \{p + q \mid p \in S_1, q \in S_2\}$$

其中，p + q 表示两个向量 p 和 q 的向量和。也就是说，若 $p = (p_x, p_y)$，$q = (q_x, q_y)$，则有

$$p + q \quad := \quad (p_x + q_x, p_y + q_y)$$

既然多边形都是平面点集，故当然也可以定义它们之间的 Minkowski 和。

为了能够利用 Minkowski 和来表述 C-障碍物，还需要借助另一个概念。对于任何点 $p = (p_x, p_y)$，定义$-p := (-p_x, -p_y)$；而对于任何集合 S，定义$-S := \{-p \mid p \in S\}$。也就是说，-S 是 S 相对于坐标原点的对称镜像。这样，就可以得出如下定理：

〖定理 13.3〗

设 $\mathcal{R}$ 为沿平面做平移运动的一个机器人，$\mathcal{P}$ 为任一障碍物。则 $\mathcal{P}$ 所对应的 C-障碍物为 $\mathcal{P} \oplus (-\mathcal{R}(0, 0))$。

〖证明〗

只需证明：$\mathcal{R}(x, y)$与 $\mathcal{P}$ 相交当且仅当$(x, y) \in \mathcal{P} \oplus (-\mathcal{R}(0, 0))$。

首先，假设 $\mathcal{R}(x, y)$与 $\mathcal{P}$ 相交，并取 $q = (q_x, q_y)$为它们的任一交点。由 $q \in \mathcal{R}(x, y)$可知，$(q_x - x,$

$q_y - y) \in \mathcal{R}(0, 0)$成立——亦即，$(-q_x + x, -q_y + y) \in -\mathcal{R}(0, 0)$成立。因为 $q \in \mathcal{P}$ 也同时成立，故有$(x, y) \in \mathcal{P} \oplus (-\mathcal{R}(0, 0))$。

反过来，设$(x, y) \in \mathcal{P} \oplus (-\mathcal{R}(0, 0))$。于是，必然存在点$(r_x, r_y) \in \mathcal{R}(0, 0)$和点$(p_x, p_y) \in \mathcal{P}$，使得$(x, y) = (p_x - r_x, p_y - r_y)$成立——亦即，有 $p_x = r_x + x$ 和 $p_y = r_y + y$ 成立。由此可知，$\mathcal{R}(x, y)$必与 $\mathcal{P}$ 相交。 ❑

因此，对于沿平面做平移运动的机器人 $\mathcal{R}$ 来说，各障碍物对应的 C-障碍物就是该障碍物与$-\mathcal{R}(0, 0)$的 Minkowski 和。（有时，$\mathcal{P} \oplus (-\mathcal{R}(0, 0))$也被称作 Minkowski 差（Minkowski difference）。实际上，数学文献中对 Minkowski 差的定义与此有别，因此我们在此避免使用这一术语。）

本节余下部分将推导出 Minkowski 和的一些有用性质，并设计一个计算 Minkowski 和的算法。

首先从一个简单的观察结论入手，它指出了 Minkowski 和的极点所具有的一个性质。

〖观察结论 13.4〗

设 $\mathcal{P}$ 和 $\mathcal{R}$ 为平面上的两个物体，设 $\mathcal{CP} := \mathcal{P} \oplus \mathcal{R}$。则 $\mathcal{CP}$ 中沿$\vec{d}$方向的极点就是 $\mathcal{P}$ 和 $\mathcal{R}$ 各自沿$\vec{d}$方向的极点之和。

这一观察结论如 图 13-16 所示。

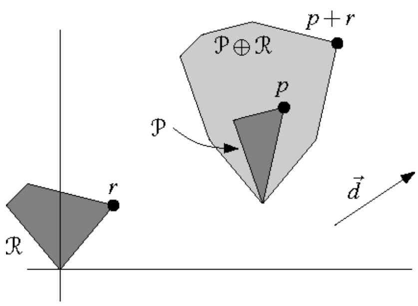


图13-16　Minkowski和的极点，必是极点之和

〖定理 13.5〗

设 $\mathcal{P}$ 和 $\mathcal{R}$ 为两个凸多边形，分别含有 n 和 m 条边。则 Minkowski 和 $\mathcal{P} \oplus \mathcal{R}$ 是一个由不超过 n + m 条边组成的凸多边形。

〖证明〗

任意两个凸集的 Minkowski 和的凸性，可以由其定义直接得证。

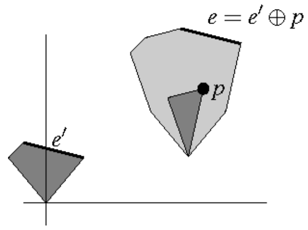


图13-17　$P \oplus R$的复杂度不超过P和R复杂度之和

下面证明：Minkowski和的复杂度是线性的。为此，如 图 13-17 所示，任取$P \oplus R$的一条边e。这条边必然是沿其外法矢的一条极边，故它必然是由P和R中沿此方向的极点（通过相加）生成的。此外，在集合P和R中，至少其一必然包含一条沿此方向的极边。我们将e记到这条边的"账"上。这样，每条边的"账"上至多有一条极边，故总的边数不会超过n + m。（若P的每条边不与R的任何边平行，则Minkowski和所含边的数目将正好是n + m。） ❑

因此，两个凸多边形的 Minkowski 和也是凸多边形，而且具有线性的复杂度。此外，任意两个 Minkowski 和的边界，只可能以某种很特别的方式相交。为便于准确阐述，需要首先明确术语。

试考虑一对平面物体o_1和o_2，设它们各由一条简单的封闭曲线围成。直观地看，物体o_1和o_2可称作一对伪圆盘（pseudodisc），如果它们的边界$\partial(o_1)$与$\partial(o_2)$至多相交于两个点（如 图 13-18 所示）。不过，在退化情况——二者的边界有一段一维的重叠部分——下，这一定义还不充分。因此，需按以下方式形式化地给出定义。物体o_1和o_2是一对伪圆盘，如果以下条件满足：$\partial(o_1)\cap \text{int}(o_2)$和$\partial(o_2)\cap \text{int}(o_1)$各自都是连通的（connected）。（这里，int(o)指代物体o的内部。）各自由一条简单的封闭曲线围成的一组物体，若其中每一对物体都是一对伪圆盘，则这些物体构成一组伪圆盘。任意一组圆盘都构成一组伪圆盘；与坐标轴平行的任意一组正方形，也构成一组伪圆盘。请注意，所谓的伪圆盘性质（pseudodisc property），指的是一对物体（的边界）之间的相交方式；谈论单个物体是否为伪圆盘，没有任何意义。

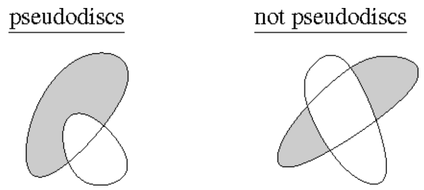


图13-18　伪圆盘性质

现在，考虑一对多边形 P 和 P'。交点 $p \in \partial P \cap \partial P'$被称为是一个边界穿越点（boundary crossing），如果∂P 在 p 处从 P'的内部转到P'的外部。多边形伪圆盘具有如下重要性质：

〖观察结论 13.6〗

任何一对多边形伪圆盘 $\mathcal{P}$ 和 $\mathcal{P}'$，最多有两个边界穿越点。

以下将证明：任意一组 Minkowski 和都是一组伪圆盘。不过，首先还需进一步观察：沿不同方向，内部互不相交的两个凸多边形各自的极点关系如何。在任一方向$\vec{d}$上，若一个多边形的极点相对于另一个的极点更远，就说“沿$\vec{d}$方向，前者比后者更加极端（being more extreme）”。例如沿 x-正方向，若一个多边形的最右侧顶点处于另一多边形最右侧顶点的右边，则前者沿该方向更加极端。

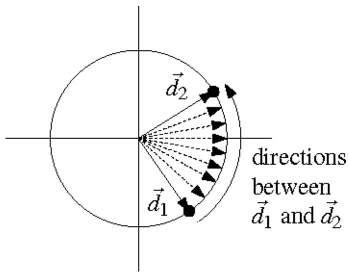


图13-19　用单位原表示所有的方向

我们需要找出所有方向上的极点。为此，如 图 13-19 所示，借助以原点为中心的一个单位圆表示所有的方向——圆上各点p所代表的方向，就是从原点到p的矢量的方向。任意两个方向$\vec{d}_1$和$\vec{d}_2$所定义的区间，就是从代表$\vec{d}_1$的点开始，沿逆时针方向到代表$\vec{d}_2$的点之间那段圆弧上各点所定义的所有方向。请注意，从$\vec{d}_1$到$\vec{d}_2$的区间，与从$\vec{d}_2$到$\vec{d}_1$的区间并不相同。由 图 13-20 可以得出如下观察结论。

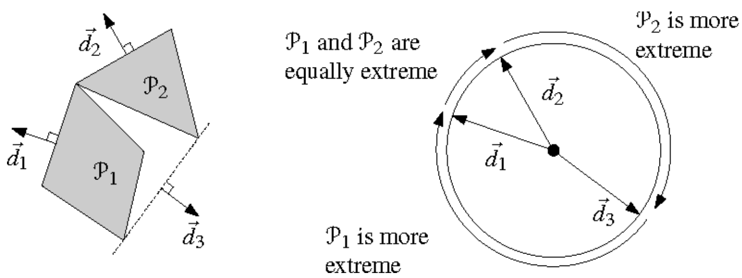


图13-20　在一段连通的区间内，沿任一方向，一个凸多边形都比另一个更加极端

〖观察结论 13.7〗

设 $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 为内部互不相交的两个凸多边形，且在方向$\vec{d}_1$和$\vec{d}_2$上 $\mathcal{P}_1$ 都要比 $\mathcal{P}_2$ 更加极端。则在从$\vec{d}_1$到$\vec{d}_2$、从$\vec{d}_2$到$\vec{d}_1$的两个区间中，必有一个区间满足：沿该区间内的任一方向，$\mathcal{P}_1$ 都要比 $\mathcal{P}_2$ 更加极端。

至此已经可以着手证明：一组 Minkowski 和必是一组伪圆盘。

〖定理 13.8〗

设 $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 为内部互不相交的两个凸多边形，$\mathcal{R}$ 为另一个凸多边形。则 Minkowski 和 $\mathcal{P}_1 \oplus \mathcal{R}$ 与 $\mathcal{P}_2 \oplus \mathcal{R}$ 必是一对伪圆盘。

〖证明〗

定义 $\mathcal{CP}_1 := \mathcal{P}_1 \oplus \mathcal{R}$，$\mathcal{CP}_2 := \mathcal{P}_2 \oplus \mathcal{R}$。根据对称性，只需证明$\partial(\mathcal{CP}_1) \cap \text{int}(\mathcal{CP}_2)$是连通的。

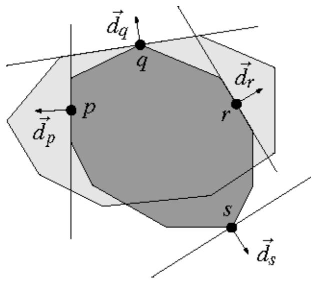


图13-21 沿着$\partial(\mathcal{CP}_1)$取四个点p、q、r和s，使得$p,r \in \text{int}(\mathcal{CP}_2)$，而$q,s \notin \text{int}(\mathcal{CP}_2)$

假设$\partial(\mathcal{CP}_1) \cap \text{int}(\mathcal{CP}_2)$不是连通的。若 图 13-21 所示，沿着$\partial(\mathcal{CP}_1)$依次取四个点p、q、r和s，使得$p,r \in \text{int}(\mathcal{CP}_2)$，而$q,s \notin \text{int}(\mathcal{CP}_2)$。考察在这四个点处的外法矢（outward normal）$\vec{d}_p$、$\vec{d}_q$、$\vec{d}_r$和$\vec{d}_s$，于是，沿$\vec{d}_p$和$\vec{d}_r$方向$\mathcal{CP}_2$ 要比$\mathcal{CP}_1$ 更为极端，而沿$\vec{d}_q$和$\vec{d}_s$方向则不然。根据 〖观察结论 13.4〗，沿$\vec{d}_p$和$\vec{d}_r$方向$\mathcal{P}_1$ 要比$\mathcal{P}_2$ 更为极端，而沿$\vec{d}_q$和$\vec{d}_s$方向$\mathcal{P}_1$ 不比$\mathcal{P}_2$ 更为极端。这与 〖观察结论 13.7〗矛盾。 ❑

上述结果如果与下面的定理结合起来，就会很有用。

〖定理 13.9〗

设 S 是一组多边形的伪圆盘，其中边的总数为 n。则它们的并集的复杂度不超过 2n。

〖证明〗

证明的思路是，按照某种原则，将并集的每一个顶点分别记到某一伪圆盘顶点的“账”上，使得每一伪圆盘顶点的账上最多只有两个顶点。如果能够做到这样，自然就说明并集的复杂度充其量不会超过 2n。

记账的原则如下。并集边界上的顶点可以分为两类：（原先某一）伪圆盘的顶点，以及（原先某两条）伪圆盘边的交点。

前一类顶点，可以直接记到自己的账上。

对交点的记账要困难一些。如 图 13-22 所示，在并集中任取一个这样的顶点v，设它是来自伪圆盘$\mathcal{P} \in S$上的边e与来自伪圆盘$\mathcal{P}' \in S$上的边e'的交点。将这两条边看成是有方向的——取

它们各自由v点进入对方内部的方向。沿着边e的方向进入P'的内部。如果这条边终止于P'的内部，就将v记到e落在P'中的这一端点的账上。反之，设e从P'的内部直接穿出。如果发生这一情况，必然会在P与P'的边界之间生成另一个交点。加上此前的交点v，共有两个交点。这就说明，e'必然不会从P的内部直接穿出——否则，就与伪圆盘的性质相悖。因此，e'必有一个端点落在P的内部，此时我们可以将v记到这一顶点的账上。

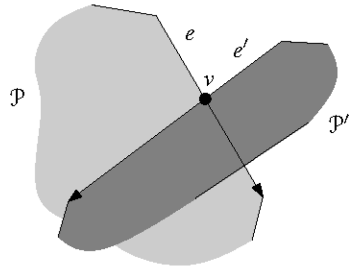


图13-22　对交点的记账

我们断言：按照上述记账规则，每一伪圆盘顶点的账上将最多记有两个顶点。对于落在并集边界上的那些顶点，这一点是显而易见的——实际上，它们只能被自己记一次账。至于其它各点，也可以按照下面的思路来证明。对于任一这类顶点，从它出发，沿着与之关联的两条边，都有可能会碰上并集的边界（如果碰到边界，该处的顶点必是这条边与另一条边的交点）。即使沿着两条边都能直接碰上并集的边界，也不过给这个顶点各记一次账；而除此之外，它不可能有其它的入账。 ❑

上述定理的证明，很大程度上依赖于一个条件——其中的伪圆盘都是多边形。但是定理本身却可以被推广至任意形状的伪圆盘。也就是说，任意一组伪圆盘的并集的复杂度，都必然线性正比于所有伪圆盘的总体复杂度。比如，由此可知：平面上任意 n 个圆盘的并集，复杂度必为 $O(n)$。不过，要证明该定理经推广后的版本，将会困难许多。

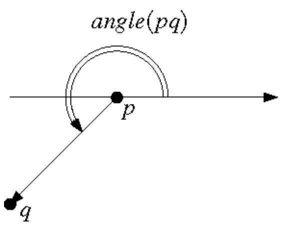


图13-23　向量$\overrightarrow{pq}$与x-轴正向所成的夹角

在重新回到运动规划问题之前，需要先给出一个算法，计算两个凸多边形P和R的Minkowski和。一个简明的算法如下。首先，对于每一对顶点$v \in P$和$w \in R$，计算出$v + w$。然后，根据这些通过加法而得到的点，构造出它们的凸包。虽然这个算法非常简单，但是由于它需要把所有可能的顶点对相加，所以一旦多边形含有很多顶点，效率就会很低。以下将给出另一种算法，而且这种算法也同样易于实现。这个算法只考虑那些在某一方向上同时为极点的顶点对——这是由〖观察结论 13.4〗

保证的——因而其运行时间是线性的。在该算法中，如 图 13-23 所示，用记号angle(pq)来表示向量$\overrightarrow{pq}$与x-轴正向所成的夹角。

```
算法 MINKOWSKISUM(P, R)
输入：由顶点 v1, ..., vn 组成的凸多边形 P，由顶点 w1, ..., wn 组成的凸多边形 R
约定.   每个多边形各自的顶点都按照逆时针方向排列
        v1 和 w1 分别为 y-坐标最小的顶点（若最小的 y-坐标相同，则取 x-坐标更小者）
输出：Minkowski 和 P ⊕ R
1.   i ← 1; j ← 1
2.   v(n+1) ← v1; w(m+1) ← w1
3.   repeat
4.     将 vi + wj 作为顶点加入 P ⊕ R
5.     if (angle(vi vi+1) < angle(wj wj+1))
6.       then i ← i+1
7.       else  if (angle(vi vi+1) > angle(wj wj+1))
8.               then j ← j+1
9.               else  i ← i+1
                       j ← j+1
10.  until (i = n+1 and j = m+1)
```

算法MINKOWSKISUM需要运行线性的时间，这是因为在**repeat**循环的每一轮中，i和j中至少有一个会增加一（这不难证明），而且一旦它们分别达到$n + 1$和$m + 1$，就不再会增加。为了证明算法所采用的顶点都是正确的，可以仿照 〖定理 13.5〗的证明方法——只需要注意到，Minkowski和的每一个顶点，必是原来在某一方向上同为极点的两个顶点之和；然后只需说明，算法中的角度测试可以保证所有的极点对都可以被检查到。

我们可以总结出如下定理：

〖定理 13.10〗

任意两个凸多边形的 Minkowski 和，都可以在 $O(n + m)$时间内构造出来，其中 n 和 m 分别为这两个多边形各自所含的顶点数。

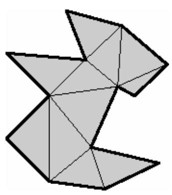


图13-24 Minkowski和遵守分配率，故通过三角剖分可以转化为凸多边形的情况

如果两个多边形之一或者全部都是非凸的，情况又将如何？只要注意到下列等式对任意三个集合 S_1、S_2 和 S3 都成立，就不难回答这个问题。

$$S_1 \oplus (S_2 \cup S3) = (S_1 \oplus S_2) \cup (S_1 \oplus S3)$$

任取一个非凸多边形$\mathcal{P}$与一个凸多边形$\mathcal{R}$，假设它们分别含有n和m个顶点，让我们来考察它们的Minkowski和。$\mathcal{P} \oplus \mathcal{R}$的复杂度是多少呢？如 图 13-24 所示，根据第 3 章的结论，我们可以将多边形$\mathcal{P}$剖分为n - 2 个三角形$\{t_1, \ldots, t_{n-2}\}$，其中n为顶点总数。根据上面的等式，可以得出：

$$\mathcal{P} \oplus \mathcal{R} = \bigcup_{i=1}^{n-2} t_i \oplus \mathcal{R}$$

既然 t_i 是三角形，而 $\mathcal{R}$ 是由 m 个顶点组成的凸多边形，故 $t_i \oplus \mathcal{R}$ 必然是由不超过 m + 3 个顶点组成的一个凸多边形。而且，由于所有三角形的内部互不相交，故这 n - 2 个 Minkowski 和必然构成一组伪圆盘。于是，它们的并集的复杂度必然线性正比于它们的复杂度总和。这就说明，$\mathcal{P} \oplus \mathcal{R}$ 的复杂度为 $O(nm)$。

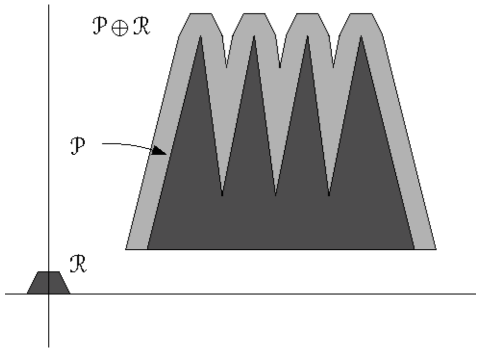


图13-25　一个非凸多边形与另一个凸多边形的Minkowski和

非凸多边形与凸多边形Minkowski和的这一上界（upper bound），在最坏情况下是紧的。为说明这一点，请考察如 图 13-25 所示的两个多边形$\mathcal{P}$和$\mathcal{R}$：$\mathcal{P}$有$\lfloor n/2 \rfloor$个指向上方的尖峰；而$\mathcal{R}$相对很小，它是正(2m - 2)边形的上半部分。这两个多边形的Minkowski和，也会有$\lfloor n/2 \rfloor$个尖峰，而且每个尖峰的顶端都有m个顶点。

为界定两个非凸多边形的Minkowski和的复杂度，可以分别对它们做三角剖分。如此可得到两组三角形：$\{t_1, \ldots, t_{n-2}\}$和$\{u_1, \ldots, u_{m-2}\}$。于是，$\mathcal{P}$和$\mathcal{R}$的Minkowski和就等于所有(t_i, u_j)对的Minkowski和的并集。每个和$t_i \oplus u_j$的复杂度都是常数。因此，$\mathcal{P} \oplus \mathcal{R}$就是(n - 2)(m - 2)个具有常数复杂度的多边形的并集。这就说明，$\mathcal{P} \oplus \mathcal{R}$的总体复杂度为$O(n^2m^2)$。同样地，这一上界在最坏情况下也是紧的——的确存在两个非凸多边形，其Minkowski和的复杂度为$\Theta(n^2m^2)$。图 13-26 就是这样一个例子。

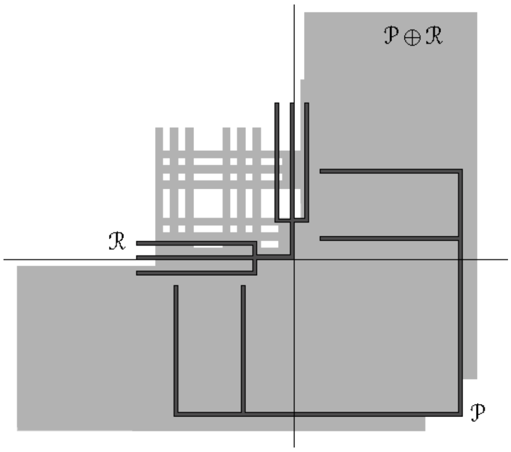


图13-26 两个非凸多边形的Minkowski和

上述关于 Minkowski 和之复杂度的结论，可以归纳为如下定理。出于完整性的考虑，定理中也给出了两个多边形同时为凸时的结论。

〖定理 13.11〗

设 $\mathcal{P}$ 和 $\mathcal{R}$ 为两个多边形，它们分别含有 n 和 m 个顶点。Minkowski 和 $\mathcal{P} \oplus \mathcal{R}$ 的复杂度上界分别如下：

(i) 若两个多边形都是凸的，则上界为 $O(n + m)$；

(ii) 若一个为凸，另一个非凸，则上界为 $O(nm)$；

(iii) 若两个多边形同时非凸，则上界为 $O(n^2m^2)$。

在最坏情况下，上面的每一上界都是紧的。

非凸多边形的 Minkowski 和，并不难构造出来——分别对两个多边形做三角剖分，然后计算出每一对三角形的 Minkowski 和，最后取它们的并集。这种方法与下一节将要介绍的另一个算法基本相同（后一算法可以针对平移式机器人构造出禁止空间），因此，我们不再对它做详细讨论。

13.4 平移式运动规划

现在可以重新回到平面上的运动规划问题。你应该还记得，我们的机器人 $\mathcal{R}$ 只能进行平移运动，而且所有的障碍物都是互不相交的多边形。上节已经说明过，任一障碍物 P_i 所对应的 C-障碍物就是 Minkowski 和 $\mathcal{P}_i \oplus (-\mathcal{R})$。另外，我们也曾看到，凸多边形的 Minkowski 和必然构成一组伪圆盘。利用这些结果，可以证明我们在运动规划问题方面的第一个主要结论——这个结论指出，对于在平面上做平移运动的机器人而言，其自由空间的复杂度是线性的。

〖定理 13.12〗

设 $\mathcal{R}$ 为一个具有常数复杂度的机器人，它可以在一组互不相交的多边形障碍物 S 之间做平移运动。则自由 C-空间 $\mathcal{C}_{free}(\mathcal{R}, S)$的复杂度为 $O(n)$，其中 n 为所有障碍物所含边的总数。

〖证明〗

首先对每个障碍物多边形做三角剖分。这样就得到了一组共$O(n)$个三角形（它们当然是凸的） 障碍物，这些障碍物的内部互不相交。此时的自由C-空间，也就是这些三角形对应的C-障碍物的并集的补集。因为机器人本身具有常数复杂度，所以每个C-障碍物的复杂度也是常数。另外，根据〖定理 13.8〗，所有这些C-障碍物必然构成一组伪圆盘。于是根据〖定理 13.9〗，它们的并集具有线性的复杂度。 ❑

剩下的任务，就是设计一个构造自由空间的具体算法。这里并不是直接去构造自由空间 $\mathcal{C}_{free}$，而是先构造出 $\mathcal{C}_{forb}$，后者的补集就是自由空间。

将由三角剖分生成的三角形分别记作 $\mathcal{P}_1, \ldots, \mathcal{P}_n$。我的目标是构造出

$$\mathcal{C}_{forb} = \bigcup_{i=1}^{n} \mathcal{CP}_i = \bigcup_{i=1}^{n} \mathcal{P}_i \oplus (-\mathcal{R}(0, 0))$$

在第 13.3 节中，我们已经知道了如何构造每一个Minkowski和$\mathcal{CP}_i$。为了构造出它们的并集，可以采用分治策略。

算法 FORBIDDENSPACE($\mathcal{CP}_1, \ldots, \mathcal{CP}_n$)

输入：一组 C-障碍物

输出：禁止空间 $\mathcal{C}_{forb} = \bigcup_{i=1}^{n} \mathcal{CP}_i$

1. **if** (n = 1)
2. **then return** $\mathcal{CP}_1$
3. **else** $\mathcal{C}^1_{forb} \leftarrow$ FORBIDDENSPACE($\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil n/2 \rceil}$)
4. $\mathcal{C}^2_{forb} \leftarrow$ FORBIDDENSPACE($\mathcal{P}_{\lceil n/2 \rceil+1}, \ldots, \mathcal{P}_n$)
5. 构造 $\mathcal{C}_{forb} = \mathcal{C}^1_{forb} \cup \mathcal{C}^2_{forb}$
6. **return** $\mathcal{C}_{forb}$

该算法的核心，是对两个平面区域构造并集的子程序。在算法中合并（由递归调用返回的区域）时（第 5 行），需要这样一个子程序。只要采用双向链接边表结构来表示这些区域，就可以通过第 2 章所介绍的叠合算法来实现这一过程。

上述结论可以归纳为如下引理：

〖引理 13.13〗

对于一个具有常数复杂度的机器人，若它在一组多边形障碍物之间做平移运动，则其对应的自由 C-空间 $\mathcal{C}_{free}$ 可以在 $O(n\log^2 n)$时间内被构造出来，其中 n 为所有障碍物所含边的总数。

〖证明〗

第 3 章中我们已知，由m个顶点组成的多边形可以在$O(m\log m)$时间内被三角剖分。（正如那一章 注释及评论部分所指出的，可用一个极其复杂的算法在$O(m)$时间内完成三角剖分。）因此，若将障碍物P_i的复杂度记作m_i，则所有障碍物的三角剖分所需要的时间将正比于

$$\sum_{i=1}^{t} m_i \log m_i \le \sum_{i=1}^{t} m_i \log n = n\log n$$

接下来，需要为剖分出来的每一个三角形，构造出对应的 C-障碍物——这总共需要线性的时间。下面，只需界定出通过 FORBIDDENSPACE 算法构造这些 C-障碍物的并集所需的时间。

由第 2 章的结论，合并计算（第 5 行）可在$O((n_1 + n_2 + k)\log(n_1 + n_2))$时间内完成，其中$n_1$、$n_2$和k分别为$\mathcal{C}^1_{forb}$、$\mathcal{C}^2_{forb}$和$\mathcal{C}^1_{forb}\cup\mathcal{C}^2_{forb}$的复杂度。由〖定理 13.12〗，自由空间的复杂度（即禁止空间的复杂度）线性正比于所有障碍物复杂度之和。这意味着，就当前的问题而言，n_1、n_2 和k都是$O(n)$，故合并计算所需总体时间为$O(n\log n)$。这样，若用T(n)表示该算法处理n个具有常数复杂度的C-障碍物所需的时间，即可得到如下关于T(n)的一个递推式：

$$T(n) = T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor) + O(n\log n)$$

其解为 $O(n\log^2 n)$。 ❑

上述定理的结论，还不是最好的。关于这方面的情况，请参见本章的 注释及评论部分。

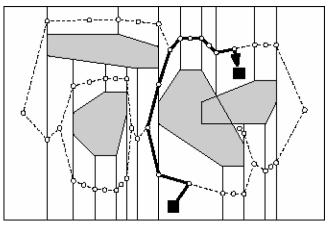


图13-27　基于自由空间的梯形图，构造可行的通路

现在，自由空间已经可以构造出来了。如 图 13-27 所示，接下来，可以完全照搬第 13.2 节所介绍的方法：构造出自由空间所对应的一幅梯形图及其对应的路线图。在为机器人任意指定了起始及终止位置之后，都可以按照下面的步骤规划出一条路径。首先，分别将起始和终止位置映射为C-空间中的点。然后，按照第 13.2 节的方法，利用梯形图和路线图，在自由空间中规划出一条联接于这

两个点之间的路径。最后，将这条路径在映射回到工作空间，即得到$\mathcal{R}$的一条路径。

在经过上面的努力之后我们所得到的结果，可以总结为如下定理：

〖定理 13.14〗

设 $\mathcal{R}$ 为一个具有常数复杂度的机器人，它可以在一组互不相交的多边形障碍物 S 之间做平移运动。在经过 $O(n\log^2 n)$期望运行时间对 S 进行预处理之后，对于任意指定的起始和终止位置，我们都可以在 $O(n)$时间内，在这两个位置之间为 $\mathcal{R}$ 规划出一条无碰撞的路径（如果的确存在这样一条路径的话），其中 n 为所有障碍物所含边的总数。

13.5 *允许旋转的运动规划

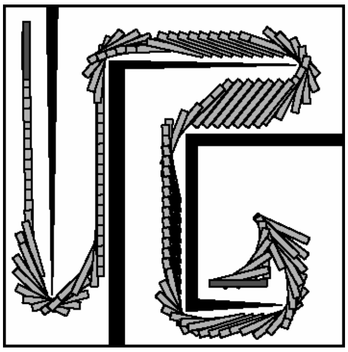


图13-28　可旋转机器人的运动规划

此前各节所讨论的机器人，都只能做平移运动。对于圆形机器人来说，这一约定并不会限制机器人的运动能力。不过，如 图 13-28 所示，要是机器人的外形又长又细，仅仅允许做平移运动往往不够。为了穿过狭窄的通道，或是在角部拐弯，它都需要改变自己的方向。本节将扼要介绍一种方法，采用这种方法，可以为既能平移也能旋转的机器人做运动规划。

设$\mathcal{R}$为一个既能平移也能旋转的多边形机器人，在它所处的平面工作空间中，有一组互不相交的多边形障碍物$\{\mathcal{P}_1, \dots, \mathcal{P}_t\}$。机器人$\mathcal{R}$具有三个自由度：两个平移自由度，一个旋转自由度。因此，$\mathcal{R}$所处的位置可以表示为三个参数：$\mathcal{R}$的参考点的x-坐标和y-坐标，再加上与其朝向对应的一个角度ϕ。与第 13.1 节的做法一样，参考点为(x, y)、旋转角为ϕ的机器人被记作$\mathcal{R}(x, y, \phi)$。

如此得到的 C-空间，就是一个三维空间 $\mathbb{R}^2 \times [0 : 360)$，该空间具有一个拓扑——(x, y, 0)和(x, y, 360)是等同的。你应该记得，障碍物 $\mathcal{P}_i$ 对应的 C-障碍物 $\mathcal{CP}_i$ 的定义如下：

$$\mathcal{CP}_i \;:=\; \{(x, y, \phi) \in \mathbb{R}^2 \in [0 : 360) \mid \mathcal{R}(x, y, \phi) \cap \mathcal{P}_i \neq \emptyset\}$$

这些 C-障碍物的形状如何？虽然这个问题很难直接回答，但是我们还是能够通过不同常数ϕ各

自对应的截片（cross-section），来获得一些认识。在每一个这样的截片上，转角都是固定的，因此我们需要处理的实际上只是一个纯粹的平移式（机器人）问题。由此可以确定每一截片的形状——它也是 Minkowski 和。更准确地说，平面 $h : \phi \equiv \phi_0$ 对 $\mathcal{CP}_i$ 的截片，就等于 $\mathcal{P}_i \oplus \mathcal{R}(0, 0, \phi_0)$。（再准确些说，这一截片是 Minkowski 和在高度ϕ_0 处的一份拷贝。）

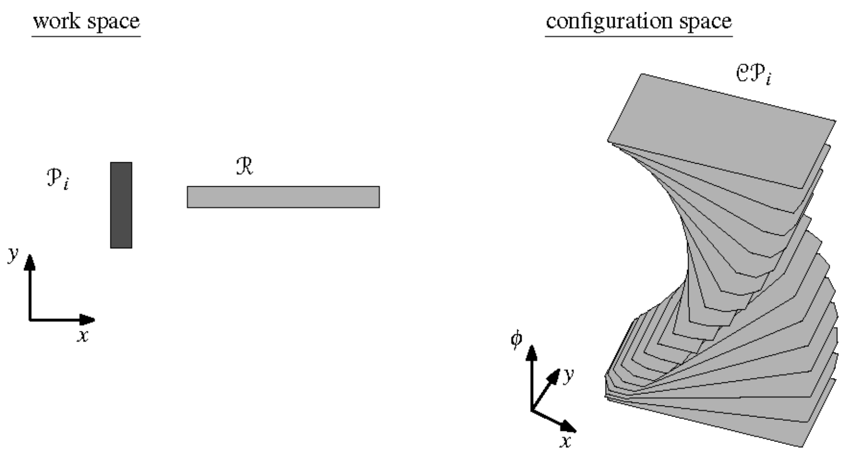


图13-29 既能平移也能旋转的机器人所对应的C-障碍物：工作空间（左），C-空间（右）

现在，想象着从$\phi = 0$ 的高度开始，用一张水平平面自下而上地扫过C-空间，直到$\phi = 360$ 的高度。在这一扫瞄过程中的任一时刻，该平面对$\mathcal{CP}_i$的截片都是一个Minkowski和。（随着平面的扫过，）Minkowski和的外形将连续变化——在$\phi = \phi_0$ 的高度，截片为$\mathcal{P}_i \oplus \mathcal{R}(0, 0, \phi_0)$；而在$\phi = \phi_0 + \varepsilon$的高度，截片为$\mathcal{P}_i \oplus \mathcal{R}(0, 0, \phi_0 + \varepsilon)$。这就意味着，$\mathcal{CP}_i$的形状就像一根扭曲的柱子（如 图 13-29 所示）。除了顶部和底部的小平面之外，扭曲柱子上的其它边和小平面都是弯曲的。

这样，我们就多少对 C-障碍物的形状有了一些认识。这些 C-障碍物的并集的补集，就是自由空间。由于 C-障碍物这种令人讨厌的外形，自由空间（的结构）将是非常复杂的——其弯曲的边界已不能用多边形来描述。此外，即使机器人的形状是凸的，自由空间的组合复杂度（combinatorial complexity）也将高达平方量级；而对于非凸的机器人，则将高达立方量级。尽管如此，我们还是能够借助此前所使用的方法，来解决（这种条件下的）运动规划问题。首先将自由空间剖分为简单的单元，然后构造一幅路线图，来为相邻单元之间的运动导航。对于任意给定的起始和终止位置，我们都可以按照下面的方法，为机器人规划出一条路径。首先将这两个位置映射为 C-空间中的两个点，然后确定它们各自所属的单元，最后规划出由三段联接而成一条路径：第一段从起始点出发，通往在路线图中位于起始单元中心的那个节点；第二段完全沿着路线图前进，一直通往位于目标单元中心的那个节点；最后一段落在目标单元内，最终达到目的地。最后的任务，就是将 C-空间中的这条路径映射回到工作空间，从而得到一条真正的运动路径。

鉴于 C-障碍物外形的复杂性，很难找到一种适当的方法来进行单元划分，而在真正实现的时候，这一点尤为棘手。因此，这里将介绍另外一种相对简单的算法。不过，正如我们将要看到的，这种

方法也有不足之处。这种方法的依据，也是我们在分析 C-障碍物形状时所得到的观察结论——具体来说就是，如果将注意力限制在 C-空间的一个水平截片上，这样的运动规划问题也就等同于一个纯粹的平移式问题。这种截片称作切片（slice）。我们的思路就是构造出有限张切片。这样，机器人的每一条路径，都可以分解为两类运动——在同一切片中的运动（即纯粹的平移运动），以及从一张切片到相邻切片的运动（即纯粹的旋转式运动）。

以下对上述构思做一形式化描述。假定共抽取z张切片。对任一整数i（$0 \le i \le z-1$），令$\phi_i = i \times (360/z)$。对每一角度$\phi_i$，分别构造出自由空间的一张切片。在每张切片上，需要解决的都是机器人$\mathcal{R}(0, 0, \phi_i)$的一个纯粹平移式问题，因此只要照搬上一节中的方法，即可构造出每张切片。于是，在每张切片上都可以得到自由空间的一幅梯形图$\mathcal{T}_i$。接下来，对每一幅$\mathcal{T}_i$分别构造出一幅路线图$\mathcal{G}_i$。按照第 13.2 节中介绍的方法，利用这些路线图，可以在各张切片上分别进行运动规划。

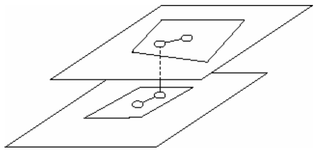


图13-30　分别在各切片上做运动规划，然后将结果串接起来

如 图 13-30 所示，剩下的任务就是将相邻的切片联接起来。更准确地说，我们要将每一对路线图$\mathcal{G}_i$和$\mathcal{G}_{i+1}$ 依次联接起来，构成整个C-空间的路线图$\mathcal{G}_{road}$。方法如下。对于每一对邻接的切片，取出它们各自对应的梯形图，采用第 2 章所介绍的算法，计算出它们的叠合。（从严格的意义上讲应该说，计算的是$\mathcal{T}_i$和$\mathcal{T}_{i+1}$ 到平面$h : \phi = 0$ 的投影之间的叠合。）这样，就可以从$\mathcal{T}_i$和$\mathcal{T}_{i+}$中找出所有相交的梯形$\Delta_1 \in \mathcal{T}_i$和$\Delta_2 \in \mathcal{T}_{i+1}$。在$\Delta_1 \cap \Delta_2$ 中任取一点(x, y, 0)。然后在$\mathcal{G}_{road}$中的(x, y, ϕ_i)及(x, y, ϕ_{i+1})处各增加一个新的节点，并引入一条弧将它们联接起来。沿着这条弧从一张切片进入下一张切片，对应于从角度ϕ_i到ϕ_{i+1}（或者反过来）的一次旋转。此外，还要将位于(x, y, ϕ_i)处的节点与位于Δ_1 中心处的节点联接起来，将位于(x, y, ϕ_{i+1})处的节点与位于Δ_2 中心处的节点联接起来。这些联接都处于同一切片中，因此它们对应于纯粹的平移运动。最后，还要以同样的方法将$\mathcal{G}_{z-1}$ 与$\mathcal{G}_0$ 联接起来。请注意，沿着图$\mathcal{G}_{road}$中各路径所对应的机器人路径，如果只限于同一切片之内，机器人只能做纯粹的平移运动；而沿着某条弧从一张切片进入另一张切片，机器人只能做纯粹的旋转运动。

一旦构造出这样一幅路线图，对任意给定的起始位置 $\mathcal{R}(x_{start}, y_{start}, \phi_{start})$和目标位置 $\mathcal{R}(x_{goal}, y_{goal}, \phi_{goal})$，都可以用它来为 $\mathcal{R}$ 进行运动规划。为此，需首先通过取整运算，确定分别与方向ϕ_{start} 和ϕ_{goal} 最接近的方向ϕ_i——如此可以找到分别与起始位置和目标位置最靠近的切片。在这两张切片中，找出起始点和目标点各自所在的梯形Δ_{start} 和Δ_{goal}。若其中有一个点落在该切片的禁止空间内，则某个梯形就可能不存在，此时报告“无法规划出路径”。否则，设 v_{start} 和 v_{goal} 分别为路线图中被放置在这两个梯形中心处的节点。我们希望在 $\mathcal{G}_{road}$ 中找出一条联接于 v_{start} 和 v_{goal} 之间的路径。若这幅图中不

存在这样的路径，则报告“无法计算出运动方案”。否则，报告出一个由五部分组成的运动方案：首先是从起点通往最近切片的一次纯粹的旋转；在该切片内通往 v_{start} 的一次纯粹平移；$\mathcal{G}_{road}$ 中联接于 v_{start} 和 v_{goal} 之间的一条路径所对应的一系列运动；在该切片（也就是与目标点最近的那张切片）上，从 v_{goal} 通往终点的一次纯粹的平移运动；最后是通往真正目标点的一次纯粹的旋转。

上面介绍的方法，是对我们求解平移式运动规划问题时所采用方法的推广。不过，这种方法存在一个主要的问题：它的正确性并不能保证。有些时候，该算法报告出来的路径可能根本就不存在。例如，即使某一起始点本身落在自由空间内，但与该起始点最靠近的那张切片却不见得。如果出现这种情况，算法会报告说“不存在通路”，而实际情况并不是这样。更加糟糕的是，即使算法能够报告出一条路径，有时路径却并不是无碰撞的。在同一张切片之内的平移运动不会出现这一问题——因为算法在这里使用的是一张准确的切片。然而从一张切片进入下一张切片的旋转运动却可能会出现问题：两张切片内各自的那个位置的确是无碰撞的，但在旋转的半途中却可能会与某个障碍物发生碰撞。只要增加切片的数目，这类问题出现的可能性就会越来越小；但（无论使用多少张切片，）我们总是不能肯定结果的正确性。后一问题尤其令人头疼——我们绝对不会希望自己耗费重金制造的机器人出现些许的磕碰闪失。

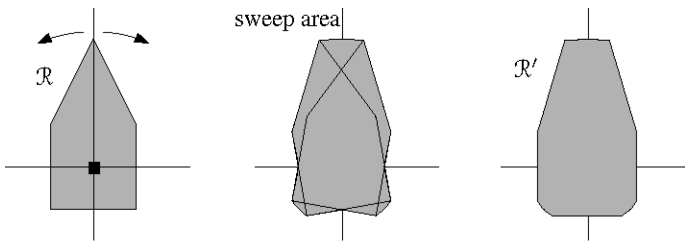


图13-31　将机器人放大

因此，我们将运用下面的技巧。首先将机器人稍做放大，然后再对放大后的机器人$\mathcal{R}'$运用上面介绍的方法。这样一来，虽然在旋转的过程中$\mathcal{R}'$可能会与障碍物发生碰撞，但原来的机器人$\mathcal{R}$却不至于发生碰撞。为了做到这一点，可以按照下面的方法来放大机器人。沿顺时针方向和逆时针方向，分别将机器人$\mathcal{R}$旋转$(180/z)^o$。在旋转的过程中，$\mathcal{R}$会在平面上扫出某一区域。我们取这一区域的凸包作为放大的机器人$\mathcal{R}'$（参见 图 13-31）。此后，我们将针对$\mathcal{R}'$而不是$\mathcal{R}$，来构造梯形图和路线图。不难证明，当通过一次纯粹的旋转运动切换到邻接的切片时，虽然$\mathcal{R}'$有可能发生碰撞，但$\mathcal{R}$却绝对不会。在将机器人放大之后，又会带来一种新的可能：算法可能会错误地报告“不存在通路”。同样地，只要切片足够多，这种错误发生的可能性也会越来越小。因此在实际应用中，当切片数目足够多时，这种方法的效果可能还算不错。

13.6 注释及评论

在很长一段时间内，运动规划问题一直都得到了人们的广泛关注，这些人既有来自计算几何（computational geometry）界的，也有来自机器人学领域的。这方面的研究成果极为丰富，本章所介绍的内容只不过是些许的皮毛。Latombe[243]就该问题的解决方法做过详尽的介绍。不过，我们所介绍的那些概念——如C-空间、自由空间及其分解、以及将几何问题转化为图搜索问题的路线图等——构成了这方面已有方法的主要基础。

这些概念的使用可以追溯到Lozano-Perez的工作 [258][259][260]。他的方法与本章介绍的方法之间存在一个重要的区别——在对自由空间进行分解时，他采用的是一种近似的方法。第 13.2 节采用了一种精确的方法，将在平面上做平移运动的机器人所对应的自由空间剖分为梯形，这一方法来自于Kedem等人最近的工作成果 [231]。Bhattacharya和Zorbas[68]提出了一种更好的算法，其运行时间已改进到$O(n\log n)$。

Schwartz和Sharir[341]提出了一种基于对自由空间进行精确分解的通用方法。该方法建立在由Collins[136]提出的一种分解方法之上。遗憾的是，该算法法所需要运行时间将随着C-空间维度的增加以双指数的速度激增。而在采用Chazelle等人 [102]提出的分解方法后，该算法可以得到改进。

在本章中我们已经看到，如果将单元分解的方法应用于在平面上做平移运动的凸机器人，就可以导出一个$O(n\log^2 n)$的算法。该算法的瓶颈，在于计算一组Minkowski和的并集。如果采用随机增量式算法 [58][280]来替换分治算法，可以将这一步的时间复杂度降至$O(n\log n)$。

三维空间中的平移式运动规划问题，可以在$O(n^2\log n)$时间解决 [22]。

针对既可平移亦可旋转的机器人，我们也简要地介绍了一个近似算法（approximate algorithm），它并不能保证所找出的路径的确存在。只要对自由空间做精确的分解，的确可以得出精确的解，不过为此需要$O(n^3)$时间 [33]。如果机器人是凸的，这个时间复杂度可以降到$O(n^2\log^2 n)$[233]。

机器人的自由空间，可以包含多个互不连通的分量。当然，机器人的运动范围必然限制在它的起始位置所属的那个连通分量之中；反之，如果需要进入另一个连通分量，就肯定要穿越禁止空间。因此，完全不必构造出整个自由空间；实际上只要构造出**单独一个**单元，就已足够了。单个单元的最坏复杂度，通常要比整个自由空间低一个数量级。可以利用这一点，来减少运动规划算法的渐进运行时间。Agarwal和Sharir的专著 [353]以及Halperin[205]的学位论文，都曾就单个单元及其与运动规划问题的联系做过详细的讨论。

从理论的角度来看，运动规划的复杂度与机器人的自由度数目呈指数关系，因此对于多自由度的机器人来说，似乎该问题是难解的。然而可以证明，只要对机器人以及障碍物的形状稍做限制，自由空间的复杂度都将是线性的 [364][365]，而在实际环境中这些条件很可能都可以满足。

单元分解法，并不是解决运动规划问题的唯一精确算法。另一种此类方法，称为收缩法（retraction method）。这种方法无需对自由空间进行分解，就能直接构造出一幅路线图。另外，还定义了一个收缩函数（retraction function）。通过这一函数，可以将自由空间中的任一点映射到路线图上的一个点。一旦做好了这些准备工作，只要将起点和终点收缩到路线图上，然后沿着路线图上的路径，就可以找出所需的路径。已经提出了各种类型的路线图和收缩函数。其中，Voronoi图就是一种很好的路线图——因为它总是与障碍物保持尽可能远的距离。如果是一个圆盘状机器人，就可以使用常规的Voronoi图；否则，必须视机器人的具体形状，使用对应于其它距离函数的Voronoi图。无论如何，总可以在$O(n\log n)$时间内构造出这样的一幅图 [255][296]——这样，就导出了一个同样能在$O(n\log n)$时间内解决平移式运动规划问题的算法。Canny[80]提出了一种非常通用的路线图方法。这种方法可以在$O(n^d\log n)$时间内解决几乎所有类型的运动规划问题，其中d为C-空间的维度（即机器人的自由度数目）。遗憾的是，这一方法不仅过于复杂，而且还存在一个缺点——在大部分时间内，机器人都需要紧贴着某个障碍物行进。人们通常都不愿采用这种运动方式。

本章主要集中讨论了精确的运动规划。实际上，还有若干的启发式方法。

例如，可以采用所谓的近似单元划分（approximate cell decomposition）[74][258][259][398]，来取代精确的划分。这类结构也常常被称为四叉树。

另一种启发式方法，是所谓的势场法（potential field method）[39][235][378]。在这种方法中，需要在C-空间中定义一个势场（potential field），使得目标点能够“吸引”机器人，而起始点“排斥”机器人。这样，机器人就会沿着势场的指向进行运动。这种方法的问题在于，机器人有可能被吸引在势场中的某一局部最低点。为了使机器人脱离这种局部的最低点，可以采用多种技术。

最近开始流行一种新的启发式算法，即所谓的概率路线图法（probabilistic road map method）[310][230]。此方法首先计算出机器人的若干随机位置，然后将这些点按照某种方式联接起来，构成自由空间的一幅路线图。借助于这幅路线图，就可以在任何起始位置和目标位置之间进行路径规划。

Minkowski和不仅在运动规划中扮演着重要的角色，在众多的其它问题中亦是如此。其中的一个例子，就是“将一个多边形放入另一个多边形”的问题 [119]。如果需要按照某种形状对布匹进行（优化的）裁减，就会遇到这种规划问题。关于Minkowski和的基本性质以及Minkowski和的计算方法，请参阅 [43]和 [197]。

本章主要讨论了机器人的路径规划问题，而没有讨论最短路径的问题。后者将是第 15 章的主题。

最后还需要注意一点。此处约定：即使在沿着路径行进的过程中机器人会与障碍物相切，我们依然认为该路径是合法的。我们常常将这类路径称为半自由路径（semi-free path）[33][340]。而不与任何障碍物相切的路径，被称为自由路径（free path）。在阅读运动规划方面的文献时，需要留意这两个词的区别。

13.7 习题

习题 13.1 设 $\mathcal{R}$ 为一只机械手，它有一个固定的底座，由 7 节杆件组成。$\mathcal{R}$ 的最后那个关节是一个柱状关节，而其余的都是旋转式关节。为了确定 $\mathcal{R}$ 的一个位置，需要哪些参数？根据你所选用的参数，与之对应的 C-空间的维度等于多少？

习题 13.2 在根据自由空间的梯形图构造出来的路线图 $\mathcal{G}_{\text{road}}$ 中，我们在每一个梯形的中心处以及每一条垂直分割线段的中点分别设置了一个节点。实际上，可以省去放在各梯形中心处的那些节点。试说明，如何对该图进行改造，使得其中只需保留落在垂直线段中点处的那些节点。（你不得因此而增加图中边的数目。）试解释查询算法的过程。

习题 13.3 试证明：$\mathcal{CP}_i$ 的具体形状，与我们为机器人 $\mathcal{R}$ 选取的参考点无关。

习题 13.4 针对下列情况，分别画出对应的 Minkowski 和 $\mathcal{P}_1 \oplus \mathcal{P}_2$：

a. $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 都是单位圆盘；

b. $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 都是单位正方形；

c. $\mathcal{P}_1$ 是单位圆盘，$\mathcal{P}_2$ 是单位正方形；

d. $\mathcal{P}_1$ 是单位正方形，$\mathcal{P}_2$ 是以(0, 0)、(1, 0)和(0, 1)为顶点的三角形。

习题 13.5 设 $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 为两个凸多边形。令 S_1 和 S_2 分别为 $\mathcal{P}_1$ 和 $\mathcal{P}_2$ 的顶点集。试证明：

$$\mathcal{P}_1 \oplus \mathcal{P}_2 = \text{CONVEXHULL}(S_1 \oplus S_2)$$

习题 13.6 试证明〖观察结论 13.4〗。

习题 13.7 〖定理 13.9〗针对一组多边形伪圆盘的并集的复杂度，给出了一个$O(n)$的上界，其中 n为所有伪圆盘包含的顶点总数。我们对准确的上界很感兴趣。

a. 假定并集的边界上包含m个来自原来各多边形的顶点。试证明：并集边界的复杂度不会超过 $2n - m$。并利用这一结果证明：并集边界的复杂度上界为 $2n - 3$。

b. 试构造一个实例说明：上述复杂度的下界（lower bound）为 $2n - 6$。

14

四叉树：非均匀网格生成

从剃须刀、电话机到电视机、计算机，几乎所有电器设备的内部，都设有某些电子线路来进行控制，使之运转。这类电路——VLSI电路、电阻器、电容器以及其它的电子元件——都分布在印刷电路板（printed circuit board）上。为了设计出各种印刷电路板，人们必须确定这些元件的摆放位置，以及如何将它们联接起来。由此引出了一系列有趣的几何问题，本章就要解决其中之一——网格生成（mesh generation，如 图 14-1 所示）。

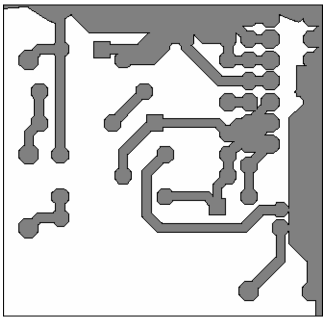


图14-1 由印刷电路板设计导出的网格生成问题

在工作过程中，印刷电路板上各个元件都会发热。为了保证电路板的正常工作，散发的热量必须控制在一定的范围以下。热量散发是否会引发问题，取决于各元件的相对位置及其联接关系，因此很难在事先进行预测。在早先的时候，人们只好先制作出一块原型电路板，然后通过试验来检查发热情况。要是试验发现温度过高，就需要调整设计方案。时至今日，已经可以对这种试验进行仿真了。得益于设计过程的高度自动化，很快就可以在计算机中建立起电路板的模型，此后的仿真计算也要比实际制作一块原型电路板快很多。另外，即使是在设计的初期，也可以通过仿真进行测试——这样，就可以尽可能早地放弃不当的设计方案。

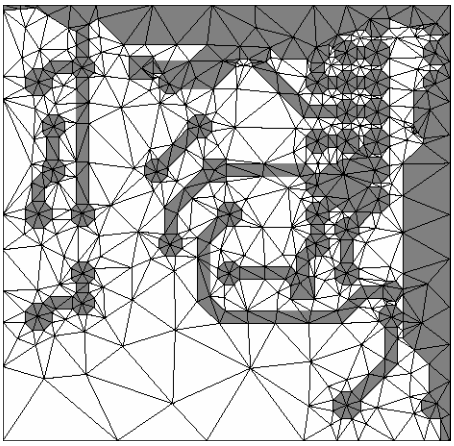


图14-2 印刷电路板三角网格的局部

印刷电路板上不同材料之间的热传导，是一个相当复杂的过程。因此，若想对电路板上的各种热过程进行仿真，就必须采用有限元法（finite element method）做近似计算。根据此类方法，首先需要将电路板划分成很多很小的子区域，称为元（element）。通常，这些元都是三角形或者四边形。我们假定，每个元自身发出的热量都是已知的。此外还要假定，邻接元之间相互影响的关系也是已知的。这样，就得出了一个大规模方程组，我们可以通过数值方法来对其进行求解。

有限元法的精度，与具体采用的网格密切相关——网格越是精细，得出的解就越是准确。然而反过来，精细的网格划分也是有代价的——随着元数目的增加，数值处理的计算复杂度将急剧攀升。因此，只有在必要的位置才应使用精细的网格。通常，这些位置就是不同材料之间的边界。还有一

点很重要，网格元不能跨越这种边界——亦即，每一网格元只能落在同一个子区域中。最后，网格元的形状也很重要——（狭长三角形之类）形状不规则的元，往往会导致数值计算的收敛速度降低。

14.1 均匀及非均匀网格

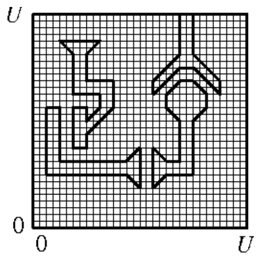


图14-3 简化的网格生成问题

此处将讨论的只是网格生成问题的一个特例。如 图 14-3 所示，输入为正方形，表示印刷电路板；其中包含若干互不相交的多边形元件。这个正方形加上其中的各个元件，有时也被称作网格的域（domain）。正方形的四个顶点，分别位于(0, 0)、(0, U)、(U, 0)和(U, U)，其中$U = 2^j$，j是一个正整数。各元件顶点的坐标，假定都是介于 0 和U之间的整数。还有一个假定条件：元件各边的方向只可能有四种选择（具体而言，每条边与x-轴的夹角只能是 0°、45°、90^o或 135°）。在很多应用问题中，这个条件都是满足的。

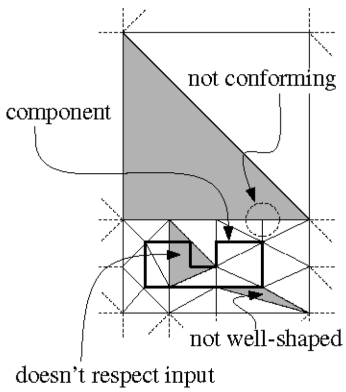


图14-4 三角形子区域划分

我们的目标，是要构造该正方形的一个三角网格（triangular mesh），也就是该正方形的一个三角形子区域划分。如 图 14-4 所示，所生成的网格必须具备如下性质：

- 该网格须是一致的（conforming）：任何三角形的顶点，不能落在其它三角形边的内部。
- 该网格须与输入相符（respect the input）：网格三角形各边的并集，须覆盖所有元件的边。
- 网格三角形须形状良好（well-shaped）：各网格三角形的内角，既不能太大，也不能过小。

具体地说，要求介于 45º 到 90º 之间。

最后，我们还希望网格只在必要的位置才足够精细。不过，这要视具体的应用环境而定。我们要求的性质如下：

- 该网格必须是非均匀的（non-uniform）：在各元件边界的附近，需要精细划分；在远离边界的地方，可以粗糙一些。

本书在前面已经讨论过三角剖分。第 3 章曾给出过对简单多边形（simple polygon）进行三角剖分的一个算法；第 9 章则给出过对点集进行三角剖分的另一个算法。后一算法可以构造Delaunay三角剖分（Delaunay triangulation）——在所有可能的三角剖分中，该三角剖分可以使最小角最大。就我们对网格三角形的角度限制而言，这种方法似乎很有用，但是这里存在两个问题。

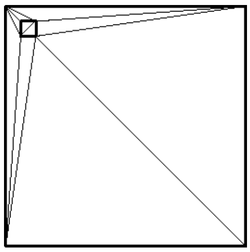


图14-5 严格由输入点生成的三角剖分

首先，各元件所有顶点的三角剖分，并不见得与元件的边界相符。即使能够相符，仍有第二个问题——某些角度可能过小。如 图 14-5，考虑一个边长 16 的正方形，其中只有一个边长为 1 的正方形元件在左上角，与大正方形的左边、上边均相距 1 个单位。于是，其Delaunay三角剖分将包含一个小于 5º的角。鉴于Delaunay三角剖分已将最小角最大化，看来似乎不可能构造出其中三角形都形状良好的网格。然而，这里却暗藏玄机——与三角剖分不同，网格中各三角形的顶点不一定非得取自输入点集。实际上，还可以通过引入新的点——称作Steiner点（Steiner point）——以获得形状良好的三角形。引入Steiner点后所生成的三角剖分，也被称作Steiner三角剖分（Steiner triangulation）。此例子中，只需在正方形每个格点上引入一个Steiner点，即可得到一个满足要求的网格——它完全由三角形组成，且每个三角形都有两个 45º的角、一个 90º的角（参见 图 14-6 左侧的网格）。

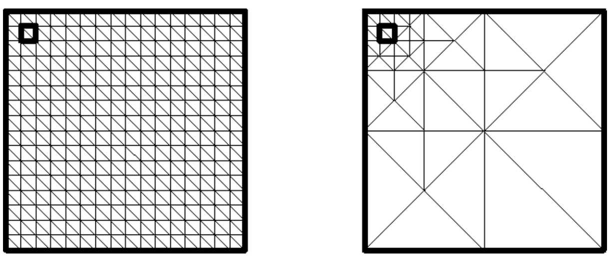


图14-6 均匀网格与非均匀网格

不幸的是，这种网格存在另一个问题：它并不是仅在输入的边界上才使用小三角形，而是处处

一样——这种网格称作均匀网格。其后果是，需要使用过多的三角形。你也许希望将正方形右下方的三角形替换为两个大三角形，然而这做不到——因为这样一来，网格将不再是一致的。尽管如此，如果随着距离左上角越来越远，我们逐渐地增大三角形的面积，那么就有可能得到一个由形状良好的三角形构成的、一致的网格（如 图 14-6 右侧所示）。这样，可以显著地减少三角形的数目——均匀网格需要 512 个三角形，而非均匀网格只需要 52 个。

14.2 点集的四叉树

接下来这一节将介绍一个基于四叉树的非均匀网格生成算法。所谓的四叉树是一棵有根的树，其中每个内部节点都有四个孩子。四叉树的每个节点v对应于一个正方形。如果v有孩子，那么它们就分别对应于v所对应正方形的四个象限——这种树结构也由此得名。亦即，各叶子所对应的正方形合起来，形成了对根节点所对应正方形的一个子区域划分。这样的子区域划分，称作四叉树划分（quadtree subdivision）。图 14-7 给出了一棵四叉树，以及与之对应的子区域划分。

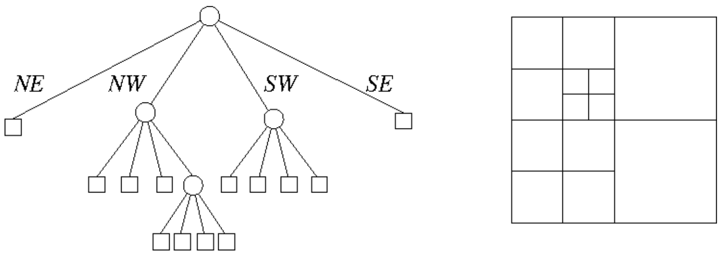


图14-7 一棵四叉树及其对应的子区域划分

根节点的四个孩子，分别标记为 NE、NW、SW 和 SE，借此表示它们各自对应的象限——NE 表示东北象限，NW 表示西北象限，等等。

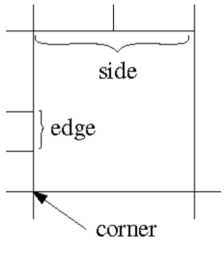


图14-8 四叉树划分中的面、侧边、边与角

在继续讲解之前，先介绍一些与四叉树划分相关的术语。在四叉树划分中，所有的面（face）都是正方形。尽管某些面（的边界上）可能有多于 4 个顶点，我们还是称之为正方形。其中位于正方形四个角上的顶点，称作角顶点（corner vertex），或简称为角（corner）。联接于同一正方形任意两个相邻角顶点之间的线段，称为该正方形的一条侧边（side）。在四叉树划分的各边中，完全落

在某个正方形边界上的边称作该正方形的边（edge）。因此，每条侧边必然包含至少一条边；当然，也可能多条。如果两个正方形共用一条边，就称它们互为邻居（neighbor）。

借助四叉树，可以存储不同类型的数据。这里将要介绍的只是其中的一种——平面上的一组点。对于这种数据，只要某个正方形中包含的点多于一个，就要递归地对其进行分割。因此对于在一个正方形 σ 中给定的一组点 P，可以定义一棵四叉树如下。设 $\sigma := [x_\sigma : x'_\sigma] \times [y_\sigma : y'_\sigma]$。

- 若 $card(P) \leq 1$，则四叉树由唯一的一匹叶子组成，其中存放了点集 P 以及正方形 σ。
- 否则，分别设 σ_{NE}、σ_{NW}、σ_{SW} 和 σ_{SE} 为 σ 的四个象限。
  令 $x_{mid} := (x_\sigma + x'_\sigma)/2$，$y_{mid} := (y_\sigma + y'_\sigma)/2$，并定义

  $P_{NE} := \{p \in P \mid p_x > x_{mid}$ 且 $p_y > y_{mid}\}$，

  $P_{NW} := \{p \in P \mid p_x \leq x_{mid}$ 且 $p_y > y_{mid}\}$，

  $P_{SW} := \{p \in P \mid p_x \leq x_{mid}$ 且 $p_y \leq y_{mid}\}$，

  $P_{SE} := \{p \in P \mid p_x > x_{mid}$ 且 $p_y \leq y_{mid}\}$。

如 图 14-9 所示，这样一棵四叉树的根节点v中存放了σ。从此，我们将存放于v处的正方形记作σ(v)。此外，v有四个孩子：

- 孩子 NE——集合 P_{NE}（即落在正方形 σ_{NE} 中各点）对应的四叉树，以它为根节点；
- 孩子 NW——集合 P_{NW}（即落在正方形 σ_{NW} 中各点）对应的四叉树，以它为根节点；
- 孩子 SW——集合 P_{SW}（即落在正方形 σ_{SW} 中各点）对应的四叉树，以它为根节点；
- 孩子 SE——集合 P_{SE}（即落在正方形 σ_{SE} 中各点）对应的四叉树，以它为根节点。

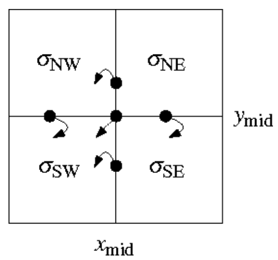


图14-9 四叉树中某个节点及其四个孩子

请注意此处定义中对“小于或等于”以及“大于”的选择，根据这种选择，每条垂直分割线都属于左侧的两个象限，而每条水平分割线都属于下方的两个象限。

四叉树中的每个节点 v，都存放有其对应的正方形 σ(v)。不过，并不一定要这样做。可以只在树的根节点处存放其对应的（整个）正方形。但是这样一来，在沿着树逐步向下移动的过程中，我们就必须不断地动态更新当前节点所对应的正方形。因此，虽然这样方法可以节省一些存储空间，却要付出其它的代价——每次对四叉树进行查询，都需要进行额外的计算。

由四叉树的递归式定义，马上就可以得到一个递归的算法：将当前的正方形划分为四个象限，同时也相应地对点集进行划分；然后，在每个象限中分别对相应的点集构造四叉树。当点集包含的点数小于 2 时，递归结束。从递归式定义中唯一不能得到的一个细节是：如何确定构造过程的起始正方形。有时，这个正方形本身就是输入的一部分。否则，可以构造出该点集的最小包围正方形（smallest enclosing square）。这很容易就能做到——在线性时间内，可以找到 x-和 y-方向的极点。

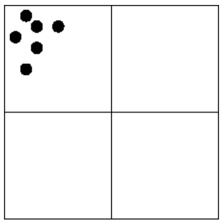


图14-10 极度失衡的四叉树

在构造四叉树过程中的每一步，某一包含多个点的正方形都会被分割为四个更小的正方形。但这并不等于说，点集本身也一定会被分割——如 图 14-10 所示，有可能原先所有的点都落在同一象限之中。因此，四叉树可能极不平衡，也无法根据其中所存点数估计其规模及深度。不过，四叉树的深度的确与其中各点之间的距离以及最初正方形的大小有关。下面的引理准确地说明了这一点：

〖引理 14.1〗

对于平面上的任何点集 P，其四叉树的深度不会超过 $\log(s/c)+\frac{3}{2}$，其中 c 是 P 中各点之间的最近距离，s 为包围 P 的初始正方形的边长。

〖证明〗

沿四叉树每下降一层，节点对应的正方形的边长就缩短一半。因此，深度为 i 的各节点所对应正方形的边长为 $s/2^i$。在每个正方形内部，任何两点之间可能的最远距离等于其对角线的长度——对深度为 i 的节点来说，也就是 $s\sqrt{2}/2^i$。因为在四叉树中，每个内部节点所对应的正方形内至少包含两个点，而且两点之间的最近距离为 c，所以内部节点的深度 i 必然满足

$$s\sqrt{2}/2^i \geq c$$

亦即

$$i \leq \log\frac{s\sqrt{2}}{c} = \log(s/c) + \frac{1}{2}$$

由于整个四叉树的深度等于所有内部节点的最大深度加一，故本引理得证。 ❑

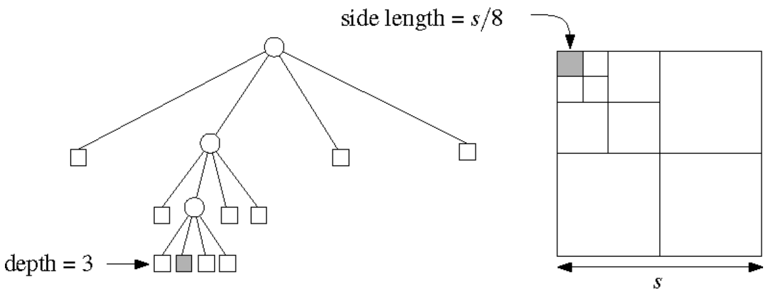


图14-11 深度为i的每个节点对应于边长为$s/2^i$的一个正方形

四叉树的规模及其构造时间，可以表示为四叉树深度以及 P 中所含点数的一个函数。

〖定理 14.2〗

存储 n 个点、深度为 d 的四叉树中，节点数目为 $O((d+1)n)$，该树可在 $O((d+1)n)$时间内构造出来。

〖证明〗

四叉树中每个内部节点都有四个孩子，故其中叶子的总数等于内部节点数目的三倍再加上一。因此，只要界定其中所含内部节点的数目即可。

在每个内部节点所对应的正方形内，都含有一个[①]或更多的点。此外，在四叉树中，处于同一深度的各节点所对应的正方形都互不相交，而且它们的并集正好覆盖了初始正方形。这就说明，在任何深度上，所有内部节点的总数不会超过n。这样就得到了关于四叉树规模的上界（upper bound）。

在递归构造算法的每一步，最耗时的任务就是将当前正方形中的点分配到四个象限中。故此，消耗在每个内部节点上的时间量，将线性正比于落在对应正方形中的点数。此前已经说明过，在树中处于同一深度各节点所对应点的总数目不会超过 n——时间上界由此得证。 ❑

邻居查找（neighbor finding）是对四叉树经常需要进行的一种操作：给定节点 v 以及一个方向（东、南、西或北），找出一个节点 v'，使得 σ(v')在指定的方向与 σ(v)近邻。通常，给定的节点都是叶子，而且往往希望找出的也是一匹叶子。这就相当于在四叉树划分中，找到与指定正方形邻接的另一个正方形。下面将要介绍的算法与此略有不同——指定的节点 v 可能是内部节点，而该算法

① 应该是“两个”。——译者

所试图给出的节点 v'，不仅满足"σ(v')从指定的方向与 σ(v)邻接"，而且 v'与 v 处于同样的深度。要是这样的节点不存在，就试图在所有邻接的正方形中找出最深的那个。要是在指定的方向上根本就没有相邻的正方形（比如 σ(v)的一条边完全包含于初始正方形的某条边中），算法将返回 **nil**。

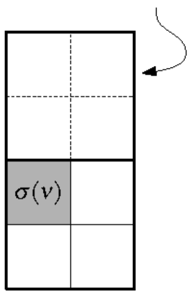


图14-12 查找指定节点的邻居

邻居查找算法的过程如下。如 图 14-12 所示，假设希望找出v在北面的邻居。要是v正好是其父节点的SE-或SW-孩子，则其北面的邻居很容易就可以找到——实际上，它要么是其父节点的NE-孩子，要么是NW-孩子。若v本身就是其父节点的NE-孩子或NW-孩子，则按照下面方法进行。我们递归地找到v的父节点在北面的邻居μ。如果μ是一个内部节点，v北面的邻居就是μ的某个孩子；否则，若μ是一匹叶子，则我们所要找的北侧邻居就是μ本身。该算法可以用伪代码描述如下：

算法 NORTHNEIGHBOR(v, *T*)

输入：四叉树 *T* 中的一个节点 v

输出：节点 v'——σ(v')从北面与 σ(v)邻接，而且 v'的深度在不超过 v 的前提下最大；
若 v'不存在，则返回 **nil**。

1. **if** (v = root(T)) **then return** nil
2. **if** (v = parent(v)的 SW-孩子) **then** **return** parent(v)的 NW-孩子
3. **if** (v = parent(v)的 SE-孩子) **then** **return** parent(v)的 NE-孩子
4. μ ← NORTHNEIGHBOR(parent(v), *T*)
5. **if** (μ = nil) 或 (μ是一匹叶子)
6.   **then return** μ
7. **else** **if** (v = parent(v)的 NW-孩子)
8.   **then return** μ的 SW-孩子
9.   **else** **return** μ的 SE-孩子

上述算法返回的并不见得是一匹叶子。如果你坚持要找出一个叶子节点，就必须从算法返回的节点开始，沿着四叉树逐层向下查找，每次都转向南侧的一个孩子。

该算法的每一次递归调用，都需要 *O*(1)时间。此外，每做一次递归，做为调用变量的节点 v 的深度就要减一。因此，整体的运行时间将线性正比于四叉树的深度。由此可以总结出如下定理：

〖定理 14.3〗

设 T 为任一深度为 d 的四叉树。按照上面的定义，任何指定的节点 v 在任何指定一侧的邻居，都可以在 $O(d+1)$时间内找到。

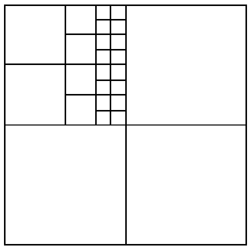


图14-13 不平衡的四叉树

我们已看到四叉树可能极不平衡。如 图 14-13 所示，面积大的正方形可能会与多个面积小的正方形相邻。在一些应用（尤其是在需要进行网格划分的应用）中，人们并不希望得到这样的结果。因此，这里要介绍四叉树的一个变种——平衡四叉树（balanced quadtree）——它没有上述问题。

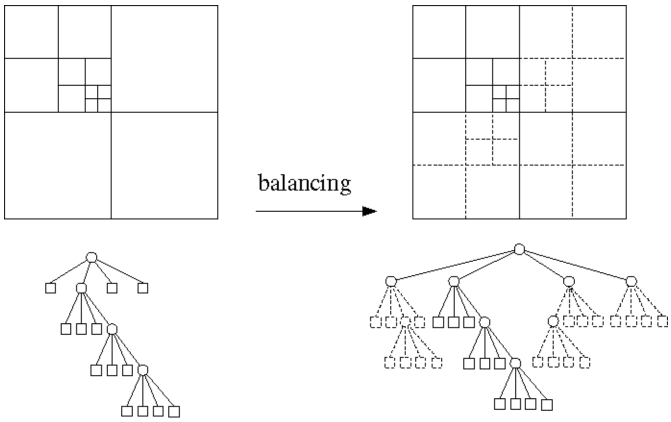


图14-14 一棵（不平衡的）四叉树及其平衡版本

在一个四叉树划分中，若任何邻接的正方形的边长相差都不超过两倍，则称之为平衡的（balanced）。若一棵四叉树所对应的子区域划分是平衡的，也称之为平衡的。因此，在一棵平衡四叉树中的任何两匹叶子，只要其各自对应的正方形是相邻的，它们的深度之差就不会大于 1。图 14-14 给出了这样的一个例子，说明了对一棵四叉树进行平衡化处理的结果。原先的子区域划分用实线表示，细分得到的部分用虚线表示。

可以按照如下算法，对一棵四叉树进行平衡化：

算法 BALANCEQUADTREE(T)

输入：四叉树 T

输出：T 经平衡化后的版本

```
1.    将T中所有的叶子组织成一个线性表L
2.    while (L非空)
3.      do 从L中摘除一匹叶子μ
4.        if (需要对σ(μ)进行分割)
5.          then 将μ做成内部节点：
                   它的四个孩子都是叶子，分别对应σ(μ)的四个象限
                 若μ中存有一个点，则
                   将该点从μ中取出，将它存入对应的新叶子中
6.               将这四匹新叶子插入L
7.               检查σ(μ)的各个邻居，看是否有某一个需要分割
                 若有某些邻居需要分割，则将它们也插入L
8.    return T
```

该算法中的两个步骤有待进一步的解释。

首先，必须判断给定的正方形σ(μ)是否需要分割。也就是说，必须检查与σ(μ)邻接的各个正方形，看看有没有那个（的边长）不到它的一半。采用上述邻居查找算法可以完成这一任务。具体的过程如下。假设我们要查找的，是从北面邻接于σ(μ)、边长不到σ(μ)一半的一个正方形。该正方形存在，当且仅当 NORTHNEIGHBOR(μ, T)所返回节点的 SW-孩子或 SE-孩子不是叶子节点。

其次，还必须判断σ(μ)的各邻居是否需要分割。同样地，这也可以借助于前面的邻居查找算法。比如，σ(μ)的北面有这样一个邻居，当且仅当 NORTHNEIGHBOR(μ, T)所返回的节点对应于一个比σ(μ)大的正方形。

至此，已经给出了一个对四叉树进行平衡化的算法。不过，为了对该平衡化算法的运行时间进行分析，需要首先回答以下问题：经过平衡化之后，四叉树的规模有何变化？图 14-14 也许会给你造成一种错觉：平衡四叉树划分（balanced quadtree subdivision）的复杂度，要远远高出未经平衡化的版本。首先，如果相邻正方形的大小相差悬殊，就需要对其中的大正方形做多次分割。其次，局部分割的效果可能会传递下去。有时，在开始的时候，某个正方形σ所有邻居的大小都符合要求，因此看起来似乎用不着分割。但是其中某些邻居本身可能需要分割，并最终导致σ需要分割。然而下面这则定理将指出，实际情况并不像乍看起来那么糟，而且平衡化处理可以高效地完成。

〖定理 14.4〗

设 $\mathcal{T}$ 为一棵包含 m 个节点的四叉树。$\mathcal{T}$ 的平衡化版本由 O(m)个节点构成，而且可以在 O((d+1)m)时间内被构造出来。

〖证明〗

首先证明节点数目的上界。将 $\mathcal{T}$ 的平衡化版本记作 $\mathcal{T}_B$。$\mathcal{T}_B$ 是在对 $\mathcal{T}$ 做了若干次细分操作之后得到的，每经过一次细分，原先的一匹叶子就会被替换为一个拥有四个叶子孩子的内部节点。我们将证明，只需要进行 8m 次这样的细分。由于每经过一次细分，（内部及叶子）节点的总数只会增加 4，这就说明 $\mathcal{T}_B$ 中所含节点的总数为 O(m)——这正是定理所指出的。

我们将 $\mathcal{T}$ 中原有各叶子所对应的正方形称作旧正方形，将属于 $\mathcal{T}_B$、但不属于 $\mathcal{T}$ 的那些节点所对应的正方形称作新正方形。在平衡化的过程中，假设我们需要对一个（无论是旧的还是新的）正方形 σ 进行分割。（σ 的四个象限的确有可能后来需要做进一步的分割，对此我们将另行统计。在目前，我们暂且将由于 σ 的分割而增加的节点数统计为 4。）稍后将证明：在围绕 σ 的八个大小相等的正方形中，至少有一个必是旧的正方形。我们将 σ 的分割记到这样一个旧正方形的“账”上。按照这种方式，每个旧正方形（等价地，即 $\mathcal{T}$ 的每个节点）最多记有八次分割。由此可知，总的分割次数不会超过 8m——这也是定理所指出的。

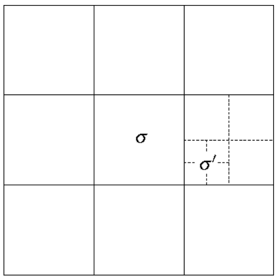


图14-15 考虑使断言不成立的最小正方形（之一）σ

接下来证明：对于在平衡化的过程中被分割的每一个正方形，围绕它的八个大小相等的正方形中至少有一个是原先已存在的。假设上述断言对某些正方形不成立。如 图 14-15 所示，在其中找出最小者，记作σ。既然σ需要做分割，它必然与一个更小的正方形邻接——准确地说，该正方形的边长不到σ的一半。考虑包含这个更小的正方形、边长正好等于σ的一半的那个正方形，记之为σ'。既然σ'包含在一个新正方形中，它自己也必然是新的。于是，在平衡化的过程中它必然也经过了分割。现在，请注意到：围绕σ'的八个大小相等的正方形必然都是新的——因为，它们要么包含在围绕σ的某个正方形之中（而σ是新的），要么包含在σ中（而根据假定，在平衡化过程中σ需要被分割）。因此，正方形σ'被分割了，而且它周围的八个正方形都不是旧的，然而它居然要比σ更小——这与σ的定义矛盾。定理的前一部分得证。

剩下只需证明：BALANCEQUADTREE 算法需要运行 $O((d+1)m)$时间。为了处理单个节点μ，只需要进行常数次的邻居查找操作，因此只需要 $O(d+1)$时间。因为每个节点最多接受一次处理，而且节点的总数为 $O(m)$，所以总的运行时间为 $O((d+1)m)$。 ❑

14.3 从四叉树到网格

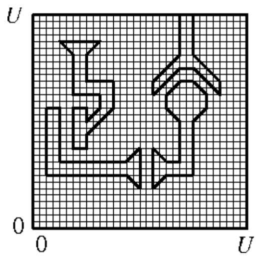


图14-16 网格生成问题

现在回到网格生成的问题。如 图 14-16 所示，你应该记得：输入为正方形$[0 : U] \times [0 : U]$，其中$U = 2^j$，j为一个正整数；该正方形内分布着若干个互不相交的多边形元件。所有多边形顶点的坐标都是整数，而且所有多边形各边的方向只有四种选择——与x-轴正向的夹角只能是 0°、45°、90^o或135^o。问题的目标是构造出该正方形（无论是元件的外部还是内部）的一张三角网格，该网格既必须是一致的，也必须与输入相符，同时其中的三角形都必须形状良好，而且分割可以是非均匀的。

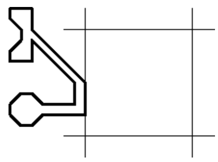


图14-17 靠近元件边界处，网格三角形应该更为精细

为得到这样一张网格，我们的思路是首先使用一个四叉树划分。在构造点集所对应四叉树的过程中，一旦正方形中包含的点数小于 2，即可终止递归构造过程。现在，需要处理的输入变成了一组多边形，因此必须重新调整递归的终止条件。如 图 14-17 所示，我们希望，在元件边界的附近网格三角形必须更为精细，因此只要正方形仍然与某条边相交，就要继续对其进行分割。更准确地说，新的递归终止条件将变成：直到正方形不再与任何元件的边界相交，或者它的边长已经缩短到一个单位，才不再继续对它进行细分。我们将每个正方形、每条边都看作闭集——这样，如果某条边与某个正方形的一条边重合，也将被视为相交。这样的终止条件，将可以保证四叉树划分是非均匀的——所有元件的边界都将被单位正方形包围；而距离各边越远，正方形也将越大。

我们断言：在得到的四叉树划分中，若某个元件的一条边与某个正方形相交于内部，则只有一种可能——它们的交为该正方形的对角线。实际上，其闭包相交的正方形必然是单位正方形，而元

件顶点的坐标全都是整数。因此，任何水平的或垂直的边都不可能与某个正方形相交于内部，而方向为 45º或 135º的边与正方形的交必然是其对角线。看起来，只要找出那些内部没有边穿过的正方形，并分别为它们添加一条对角线，似乎就可以得到一个令人满意的网格。如此生成的三角形的确与输入相符，它们的形状也很好，而且网格也是非均匀的。然而遗憾的是，这样的网格并不是一致的。为了解决这个缺陷，在对每个正方形做三角剖分的时候，还需要考虑到落在正方形边上的子区域划分顶点。不过，如 图 14-18 所示，这又会引起另一个问题：要是在某个正方形的一条边上落有很多的顶点，如此生成的三角形不见得都是形状良好的。

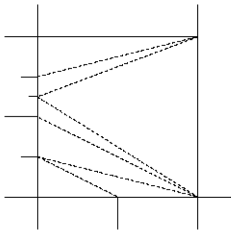


图14-18 多个点落在同一条边上时，将生成形状不良的三角形

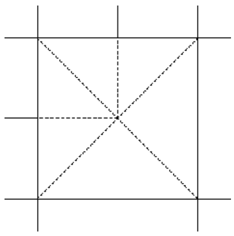


图14-19 在正方形的中心引入Steiner点

为了避免这些问题，在进行三角剖分之前，需要对四叉树划分做平衡化处理。一旦得到了一个平衡四叉树划分，很容易就可以按照如下方法，生成一张由形状良好的三角形构成的网格。对于那些侧边内部不含顶点（而且还没有被某一元件的边三角剖分过）的正方形，可以直接给它添加一条对角线。既然此时的子区域划分是平衡的，在其余各正方形的每一侧边的内部，至多只会有一个顶点。此外，若存在这样的一个顶点，则它必然落在该侧边的中点处。因此，如 图 14-19 所示，只要在该正方形的中心加入一个Steiner点，并将它与正方形边界上的所有顶点相联，得到的每个三角形就将只含有 45º或 90º的内角。

归纳起来，可得如下网格生成算法：

算法 GENERATEMESH(S)

输入： 正方形[0 : U] × [0 : U]内的一组元件 S，元件的性质如本节开头部分所述。

输出： 三角网格 *M*
（要求 *M* 是一致的，与输入相符，完全由形状良好的三角形组成，而且是非均匀的）

1. 在正方形[0 : U] × [0 : U]内，构造点集 S 的一棵四叉树 *T*

这里采用的（递归）终止条件是：
只要正方形的边长大于 1，且其闭包与某个元件的边界相交，就对其进行分割

2. $\mathcal{T}$ ← BALANCEQUADTREE($\mathcal{T}$)
3. 针对 $\mathcal{T}$ 所对应的四叉树划分 M，构造一个双向链接边表结构
4. **for** (M 中的每一张面 σ)
5. **do if** (某个元件的一条边与 σ 相交于 σ 的内部)
6. **then** 将它们的交集（实际上是 σ 的一条对角线）做为一条边插入到 M 中
7. **else if** (σ 只在四个角上有（子区域划分的）顶点)
8. **then** 将 σ 的这条对角线作为一条边插入到 M 中
9. **else** 在 σ 的中心增加一个 Steiner 点
将 σ 边界上的每个顶点与该 Steiner 点相联
相应地更新 M
10. **return** M

由算法 GENERATEMESH 生成的网格，是以双向链接边表（doubly-connected edge list）的形式表示的。至于如何对双向链接边表进行处理的细节（比如，对于给定的一棵四叉树，如何构造与之对应的双向链接边表结构），不再赘述。

由上述算法构造的网格所具有的性质，可以归纳为如下定理：

〖定理 14.5〗

设 S 为正方形[0 : U] × [0 : U]内一组互不相交的多边形元件，元件的性质如本节开头部分所述。则必然存在与该输入对应的一张非均匀的三角网格，而且该网格是一致的，与输入相符，而且完全由形状良好的三角形构成。其中三角形的数目为 O(p(S)logU)，这里的 p(S)为 S 中各元件的周长总和；这样的一张网格可以在 $O(\text{p(S)}\log^2\text{U})$时间内构造出来。

〖证明〗

根据前面的讨论，马上就可以得出该网格所具有的性质——即，它必然是非均匀的、一致的，不仅与输入相符，而且完全由形状良好的三角形构成。需要进一步证明的，只是该网格规模及其处理时间的上界。

网格构造的过程分为三步：首先构造一个四叉树划分，然后使之平衡化，最后再对其做三角剖分。

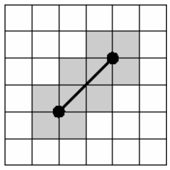


图14-20 与线段相交的单元数上界，取决于线段的长度

为界定由第一步生成的四叉树划分的规模，需要借助下面的观察结论。如 图 14-20 所示，试考虑由单位尺寸的单元所构成的格子（grid）。对任何一条长度为l的线段，其闭包与该线段相交的单元总数，不会超过 $4 + 3l/\sqrt{2}$。于是，其闭包与至少一个元件的一条边相交的单元总数，应该为$O(p(S))$。显然，对于由更大单元构成的格子，这一点依然成立。因此，在四叉树的任一固定深度，所有内部节点的总数必是$O(p(s))$。既然在单元缩小到单位尺寸时，我们将必定不再分割，因此四叉树的深度应该是$O(\log U)$。由此可知，四叉树所含节点的总数（亦即其对应的子区域划分的复杂度）应该是$O(p(s)\log U)$。

由 〖定理 14.4〗可知：从渐进意义来衡量，在经平衡化处理之后，四叉树划分的复杂度并不会增加。网格生成算法的第三步（即对平衡四叉树划分中的各正方形做三角剖分）亦是如此——这是因为，每个单元最多只能被分割成常数个三角形。总之，最终所得网格中的三角形数目，将线性正比于第一步所生成四叉树的复杂度，我们已经证明这一复杂度为$O(p(s)\log U)$。

最后一点，是要证明构造时间的上界。第一步计算时间，主要取决于四叉树的递归构造算法。消耗在每个节点上的这部分时间，将线性正比于该节点的闭包与元件各边相交的总数。此前已经说明，在任一固定的深度，所有节点与各边相交的总数为$O(p(S))$。因此，第一步所需的时间为$O(p(s)\log U)$。根据 〖定理 14.4〗，为进行平衡化处理，需要再引入一个$O(\log U)$因子。无论是根据一棵给定的四叉树构造一个双向链接边表结构,还是对平衡四叉树划分做三角剖分，都可以在同样量级的时间内完成。定理所述的处理时间，由此得证。 ❑

14.4 注释及评论

四叉树是最早用于处理高维数据的数据结构之一。这种结构是由Finkel和Bentley[177]在 1974 年提出的。此后，有数以百计的论文对四叉树做过讨论。Samet的综述以及专著 [332][333][334][335]，以及 [14]中Aluru撰写的一章，对四叉树的各种变型及其应用有详实的介绍。

网格生成只不过是四叉树的应用之一。在计算机图形学、图像分析、地理信息系统等众多领域，这一结构都有应用。通常，这种结构都是被用来支持区域查找，不过对于其它的操作，它同样也能支持。从理论的角度来看，四叉树并不是解决区域查找问题的好办法——因为一般都不能证明出一个低于线性的查询时间上界。对于各种区域查找问题，还有其它的解决方法，这方面的情况请参见第 5、10 以及 16 章。在实践中，四叉树的性能看起来往往还不算差。四叉树还可以被用来解决隐藏面消除、光线跟踪、中轴转换、光栅化图的叠合、最近邻查找（nearest neighbor search）等问题。

四叉树可以轻而易举地被推广至三维情况，在那里，这种结构称为八叉树（octree）。

无论是平面的还是三维空间的网格生成，都是众多领域之中的一个重要问题，也因此被详尽地研究过。Bern和Eppstein的综述 [62]以及Ho-Le的综述 [215]，讨论了特定条件下网格生成问题的有关结果，为该领域的初学者提供了很好的指导。我们在此对其中的几个结果做一扼要介绍。

我们可以区分所谓结构化的（structured）和未结构化的（unstructured）网格。结构化的网格通常都是（变形后的）格子；未结构化的网格通常就是三角剖分。我们的讨论限制于未结构化的网格。另外，我们的注意力主要集中于待网格化的域是二维多边形区域的情况。在大多数的应用中，对网格的要求都类似于本章所做的限定。在实践中，通常都要求网格是一致的（“conforming”一词有时也可以换成“consistent”），并要求与输入相符。此外，某种意义上的“形状良好性”也很重要。通常，这意味着三角形必须满足下面两个准则之一或者全部：(i) 不允许出现小角度——亦即，每个角度都不得小于某个给定的（不是很小的）常数θ。当然，这个常数不可能大于输入域本身的最小角度——因为在生成的网格中不能排除掉任何输入的角度。(ii) 不允许出现钝角（obtuse angle）——亦即，任何角度都不能大于 90°。本章所讨论过的例子，要求三角形同时满足这两个条件，而且取θ = 45°。我们的目标，常常是在给定的条件下，使构成网格的单元数目最少。这就意味着某种类型的非均匀性——只有在必要的时候，才使用小角度。

首先考虑在不允许出现小角度的前提下，如何使网格中的三角形最少。在这一前提下，需要多少个三角形才能对一个多边形域做网格剖分呢？这不仅取决于域中所含的顶点数目，也和域的形状有关。为说明这一点，可以引入一个指标。该指标与三角形的最小角密切相关，称作三角形的纵横比（aspect ratio）。三角形的纵横比，就是其最长边与其高度（height）的比——所谓三角形的高度，就是其最长边与该边的对顶顶点之间的欧氏距离。如果一个三角形的最小角为θ，则其纵横比将介于1/sinθ与 2/sinθ之间。接下来，再考虑这样的一个矩形区域：其短边的长度为 1，长边的长度为A。假定要求最小角不得小于 30°。这就意味着，网格中所有三角形的纵横比都不得超过 $2/\sin 30^o = 4$。此外，该区域中所有三角形的高度也不得超过 1。于是，每个三角形的面积都是$O(1)$。因为该区域总的面积为A，所以任何满足要求的网格都至少需要Ω(A)个三角形。Bern等人 [64]提出了一种基于四叉树的方法，使得生成的三角形数目是渐进最优的。本章所介绍的方法，就是基于他们的技术。

针对“网格中所有的三角形都必须是非钝的（non-obtuse）”这一限制条件，相关的研究结果表明，对于任何给定的多边形区域，总是可以构造出一个网格，使得其中三角形的数目只取决于该区域所含顶点数目。更加准确地说，Bern和Eppstein[63]证明了：包含n个顶点的任何多边形区域，都有一张由$O(n^2)$个非钝三角形构成的网格。就在最近，Bern等人 [67]又将这一结果改进到$O(n)$。

Melissaratos和Souvaine[278]将Bern等人的方法做了推广，使之能够构造出既不包含小角度也不出现钝角三角形的网格。在如此构造出来的网格中，三角形的数目仍然不会超过最优值的常数倍。

将三角形最少化，并不总是网格生成算法的目标。另一重要的方面，则是要能够控制网格的密度——这样，人们即可在感兴趣处使用稠密网格，在其它地方则使用粗糙网格。Chew[117]针对这种要求的网格生成问题做过研究。他给出了一个网格生成算法，该算法允许用户通过定义一个函数，来判断网格中的某个三角形是否已经足够精细。该算法所生成三角形的角度介于 30°和 120°之间。此项研究成果的另一个优点是，该算法不仅能够处理平面区域，而且可以处理曲面上的区域。

14.5 习题

习题 14.1　在图 14-6 中，分别给出了同一个区域的均匀网格和非均匀网格，这个正方形区域的边长为 16，其左上角处有一个单位正方形。试考虑对边长为 $U = 2^{j}$（j为正整数）的一个更大正方形的类似网格剖分。分别对两种（均匀和非均匀）网格，用j来表示其中三角形的数目。

习题 14.2　假设需要在宽度为 1、长度为 $k > 1$ 的一个矩形中构造三角网格。在区域的侧边上不得加入 Steiner 点，但在矩形的内部却可以。同时假定，所有三角形的角度都必须介于 30° 到 90° 之间。按照这样的要求，是否总是能够生成一张三角网格？假设对于某一特定的输入的确可以生成这样一张网格，至少需要加入多少个 Steiner 点呢？

习题 14.3　利用本章介绍的算法所生成的每个三角形都是非钝的——亦即，其中不存在超过 90° 的角度。试证明：对平面上的任何一个点集 P，若其三角剖分中不包含任何钝角三角形，则该三角剖分必是 P 的 Delaunay 三角剖分。

习题 14.4　设 P 为三维空间中的一个点集。试给出一个算法，构造 P 的一棵八叉树（octree）。（所谓八叉树，就是四叉树在三维空间中的推广。）

习题 14.5　正方形内一组（实数坐标的）点的四叉树，规模可以从 $O((d+1)n)$降低到 $O(n)$，其中 d 为四叉树的深度。改进的构思是，只要某个节点 v 的（四个）孩子中只有一个存有点，就将节点 v 删除。具体的删除方法是：将原先由 v 的父节点指向 v 的指针，替换为一个从 v 的父节点指向 v 唯一存有点的那个孩子。试证明：这样得到的树，具有线性的规模。$O((d+1)n)$的构造时间是否也可以改进？

习题 14.6　在本章中，如果四叉树划分中任何两个相邻正方形的边长相差不超过两倍，就称作平衡四叉树。为了将用以平衡四叉树的附加节点的数目降低一个常数因子，我们可以降低平衡的标准——比如说，只要求相邻正方形的边长相差不超过四倍。如此得到的，就是所谓的弱平衡四叉树（weakly balanced quadtree）。我们是否仍然可以由这样的一个子区域划分，得到一个一致的网格，使得其中所有的角度都介于 45° 到 90° 之间，而且每个正方形中只有 $O(1)$个三角形呢？

习题 14.7　假定更为严格地设置平衡的标准——相邻正方形的边长即使相差不超过两倍，也不算

平衡；只有其边长完全相等，才算平衡。在这样一棵强平衡四叉树中，节点的数目仍然会线性正比于普通的平衡四叉树吗？若不是，关于这个数你能做出什么断言？

习题 14.8 构造平衡四叉树的算法，分成两个阶段：首先，构造一棵普通的四叉树，接着在后处理阶段对其做平衡化处理。实际上，不用首先构造出一个非平衡的版本，就可以直接构造出一棵平衡四叉树。为此，在四叉树的构造过程中，需要借助双向链接边表结构，动态地维护当前的四叉树划分；在每个正方形被分割之后，我们都要进行检查，判断其邻居是否需要分割。试详细描述这一算法，并对其运行时间做一分析。

习题 14.9 GENERATEMESH 算法中有一步，是对某一给定的四叉树，构造一个双向链接边表，以表示其对应的四叉树划分。试描述这一步所采用的算法，并对其运行时间做一分析。

习题 14.10 也可以使用四叉树来存储子区域划分，以支持有效的点定位查询。其构思是：取包围该子区域划分的一个正方形，不断进行分割。分割的终止条件是：所有叶子对应的正方形中最多只包含一个顶点，而且只包含与该顶点相关联的边；或者其中没有任何顶点，而且至多只包含一条边。

a. 因为每个顶点都可能与多条边相关联,所以我们需要为四叉树的叶子配备附加的数据结构，以存放多个顶点。你会选用哪种数据结构呢?

b. 试详细地描述这种点定位数据结构的构造算法，并对其运行时间做一分析。

c. 试详细描述对应的查找算法，并对其运行时间做一分析。

习题 14.11 四叉树经常被用来存储像素图像。在这种情况下，初始正方形的大小正好与图像相等（假定为 $2^k \times 2^k$ 的格子，k 为整数）。只要正方形中各像素的亮度不完全相等，就将它分割为（四个）子正方形。

试证明这种四叉树划分的复杂度上界。

提示：该上界类似于我们所证明的四叉树网格的上界。

习题 14.12 假设我们已经获得了两幅像素图像 I_1 和 I_2 各自对应的四叉树（参加上题）。两幅图像的大小都是 $2^k \times 2^k$，而且像素亮度的取值只有两种：0 或 1。试给出一个算法，对这两幅图像进行布尔运算——亦即，该算法能够计算 $I_1 \vee I_2$ 和 $I_1 \wedge I_2$。（这里，$I_1 \vee I_2$ 也是相同尺寸的一幅图像，其中像素(i, j)的亮度为 1 当且仅当(i, j)在图像 I_1 中亮度为 1，或者在图像 I_2 中亮度为 1。图像 $I_1 \wedge I_2$ 的定义与此类似。）

习题 14.13 四叉树也可以用以支持区域查找。试描述一个算法，利用点集 P 对应的四叉树，对子区域 R 进行查找。就 R 是一个矩形的情况，以及 R 是由垂线界定的一张半平面的情况，分别对最坏情况下的查询时间做一分析。

习题 14.14 本章对存储平面点集的四叉树做了讨论。第 5 章也曾讨论过另外两种可存储平面点集的数据结构：kd-树和区域树。试对这三种数据结构做一比较，就其各自的优、缺点做一讨论。

15

可见性图：求最短路径

第 13 章曾经讨论过，如何在给定的起始位置和终止位置之间，为机器人规划出一条路径。只要这样的路径存在，我们所给出的算法总是能够把它找出来；然而关于这条路径的质量如何，我们却没有给出任何结论。这条路径可能要兜一个大圈，或者要转很多不必要的弯。在实际的环境中，我们更希望能够找出一条好的路径，而不仅仅只是其中任意的一条。

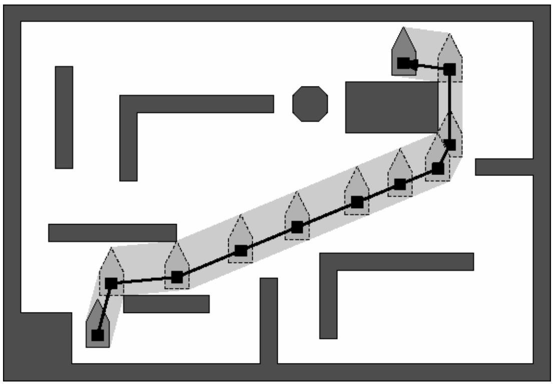


图15-1 一条最短路径

什么样的路径才是好的？这要取决于具体的机器人。一般而言，路径越长，机器人就需要更长的时间才能到达终点。对于在厂房中工作的机器人来说，路径越长，它在单位时间内能够搬运的物品就会越少，从而导致生产率的下降。因此，我们更希望采用短的路径。常常还有其它的一些因素需要考虑。例如，有的机器人只能沿着直线运动；每变更一次前进的方向，都需要减速、制动，然后再转动——这样，路径中的每一次转向，都会造成一定的延迟。对于这类机器人，就不仅需要考虑路径的长度，还应该考虑到沿途的转向次数。不过，本章将暂不考虑这些因素；这里讨论的问题仅仅是：如何为沿平面运动的机器人规划出一条欧氏最短路径（Euclidean shortest path）。

15.1 点机器人的最短路径

像第 13 章一样，首先考虑这样一种情况：一个点机器人，在平面上移动于一组互不相交的简单多边形（simple polygon）S之间。S中的每个多边形都被称作一个障碍物（obstacle），它们所含边的总数记作n。障碍物都是开集——也就是说，允许机器人与它们相切。给定一个起始位置p_{start}和一个终止位置p_{goal}，假定它们都属于自由空间（free space）。我们的任务是，找出由p_{start}通往p_{goal}的一条最短无冲突路径——即与任何障碍物内部都不相交的一条最短路径。请注意，只能说“一条最短的”，而不能说“最短的那条”，因为这样的路径不见得是唯一的。为了保证这样的路径至少存在一条，很重要的就是要把所有障碍物都看作开集；否则，倘若它们是闭集，那么这样的最短路径通常都不会存在（除非最短路径只由单独的一条直线段组成）——因为，对于任何一条路径，我们都可以将它朝某个障碍物靠拢，从而得到一条更短的路径。

让我们对第 13 章所介绍的算法做一简要回顾。如 图 15-2 所示，先构造出自由C-空间（free configuration space）$\mathcal{C}_{free}$的一个梯形图$\mathcal{T}(\mathcal{C}_{free})$。对于任何点机器人，$\mathcal{C}_{free}$就是各障碍物之间的空间，因此这种情况非常简单。这里的一个关键构思是：将原先包含无数条路径、连续的工作空间，替换为一张离散的路线图（road map）$\mathcal{G}_{road}$。我们以前所使用的路线图是一张平面图，其中，对应于$\mathcal{T}(\mathcal{C}_{free})$中各个梯形（trapezoid）的中心以及介于毗邻梯形之间的各垂线中点，分别有一个节点。每个梯形中心处的节点，与该梯形（垂直）边界上的每个节点相联。一旦确定了机器人的起点和终点各自所

在的梯形，就可以通过广度优先搜索（breadth-first search），在路线图中找到一条路径，将这两个梯形中点所对应的节点联接起来。

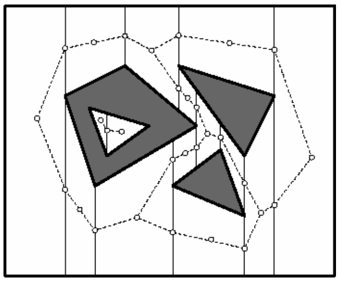


图15-2 基于自由空间的梯形图，构造可行的通路

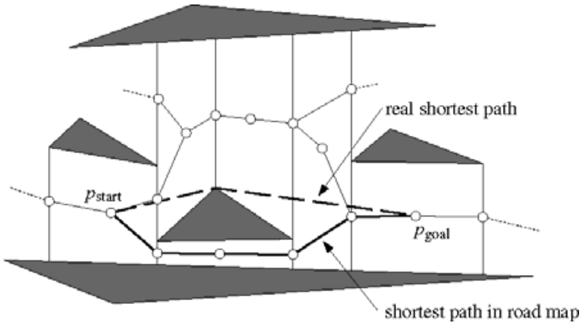


图15-3 最短路径不见得是路线图的子图：实线为沿路线图的最短路径，虚线为真正的最短路径

因为使用的是广度优先搜索，所以必然会从$\mathcal{G}_{road}$中找出使用弧最少的一条边。但它并不见得就是最短的路径，因为有些弧联接的节点可能相距很远，而另一些则可能很近。一种显而易见的改进方法是：根据每条弧所联接节点之间的欧氏距离，赋予一个权重；然后利用Dijkstra算法（Dijkstra algorithm）之类的图搜索算法，在该带权图中找出一条最短路径。这的确能够缩短路径的长度，然而我们得到的仍然不是最短路径。这一点可以从 图 15-3 中看出：若必须沿着路线图，则从p_{start}到p_{goal}的最短路径就会从三角形的下面穿过；而真正的最短路径，却是从其上方穿过。为了保证沿图中各弧的最短路径是真正的最短路径，我们需要另一种路线图。

我们来看看，关于最短路径的形状，可以有什么结论。如 图 15-4 所示，考虑从p_{start}通往p_{goal}的某条路径。将这条路径想象为一根有弹性的橡皮筋：两端分别被固定在起点和终点处，其形状被强制为与路径一致。一旦松开这根橡皮筋，它就会开始收缩，并且尽可能变短，直到碰到障碍物才停下来。沿着此时新的路径，有些段是障碍物的部分边界，而其它部分则是穿越开放空间的直线段。下面的引理，对这一观察结果做了准确的描述。其中使用到多边形路径中内部顶点（inner vertex）的概念——即该路径上除起点、终点之外的任何顶点。

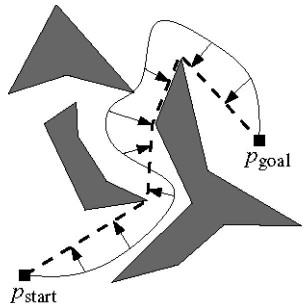


图15-4 最短通路

〖引理 15.1〗

穿行于一组互不相交的多边形障碍物 S 之间、从 p_{start} 通往 p_{goal} 的任何一条最短路径，都是一条多边形路径，其中所有的内部顶点都是 S 的顶点。

〖证明〗

假设引理不成立，即存在某条最短路径τ不是多边形链。所有障碍物都是多边形，这就意味着在τ上存在这样的一个点p：p落在自由空间的内部，而且经过p的任何一条线段都不会属于τ。如图 15-5 所示，既然p落在自由空间的内部，就必然存在一个以p为中心、半径为正数的圆盘，完全落在自由空间中。

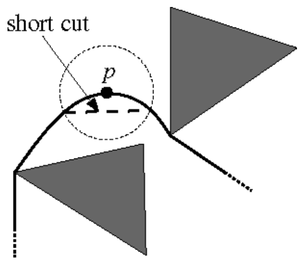


图15-5 在自由空间内部，以任一点p为中心，存在一个正半径的圆盘

然而根据假设，τ 穿越该圆盘的那一段不是直线段。考虑 τ 进入和离开该圆盘的两个位置，只要用联接于这两个点之间的线段替换这段路径，就可以使整条路径的长度缩短。这与 τ 的最优性矛盾——因为，任何最短路径必然也是局部最短的（locally shortest），也就是说，该路径上任何两点 q 和 r 之间的那段（子）路径，必然是由 q 通往 r 的最短路径。

现在，考察τ上的任一顶点v。它不可能落在自由空间的内部。否则，必然存在以v[①]为中心、完全包含在自由空间中的一个圆盘，于是就可以将τ穿越该圆盘的那段子路径替换为一条直线段——由于原先的那段子路径至少在v处不是直的，所以替换后路径的长度必然缩短。类似地，v也不可能落在任何一条障碍物边的相对内部。否则，必然存在以v为中心的某个圆盘，它的一半

① 原书误作 p。——译者

属于自由空间——这同样意味着，可以将穿越圆盘的那段子路径替换为一条直线段（从而缩短路径的长度）。这样，v可能的位置只能是障碍物的顶点。 ❑

最短路径的上述特性确定之后，即可构造出一张路线图，并借助它找到最短路径。这张路线图称作 S 的可见性图（visibility graph），记作 $\mathcal{G}_{vis}(S)$。其中每一节点分别对应于 S 中的顶点；若顶点 v 与 w 可以相互看见，则在它们对应的节点之间引入一条弧。这里所谓的看见（see），指的是线段 $\overline{vw}$不与 S 中任何障碍物的内部相交。若一对顶点可以相互看见，就称它们为相互可见的（visible），而联接它们的线段则称作一条可见边（visibility edge）。请注意，障碍物任何一条边的两个端点，总是相互可见的。因此，由所有障碍物的边所构成的集合，必是 $\mathcal{G}_{vis}(S)$边集的一个子集。

根据〖引理 15.1〗，在组成最短路径的各条线段中，除第一条和最后一条外都是可见边。为了使这两条边也成为可见边，我们将起点、终点也做为顶点加入到S中——也就是说，我们考虑的是集合$S^* := S \cup \{p_{start}, p_{goal}\}$的可见性图。根据定义，$\mathcal{G}_{vis}(S^*)$的每条弧，都联接于彼此可见的一对顶点（现在可能是$p_{start}$或$p_{goal}$）之间。由此得出下列推论。

〖**推论** 15.2〗

穿行于一组互不相交的多边形障碍物 S 之间、从 p_{start} 通往 p_{goal} 的任何一条最短路径，必然是由可见性图 $\mathcal{G}_{vis}(S^*)$中的若干条弧联接而成的，其中 $S^* := S \cup \{p_{start}, p_{goal}\}$。

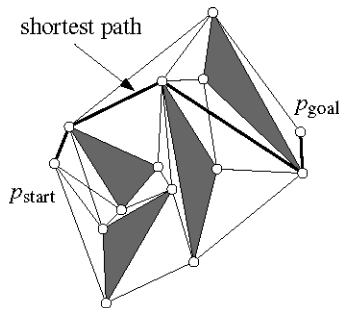


图15-6 p_{start}与p_{goal}之间任一最短路径，都由可见性图$\mathcal{G}_{vis}(S^*)$中若干条弧联接而成

利用如下算法，可在 p_{start} 与 p_{goal} 之间规划出一条最短路径：

算法 SHORTESTPATH(S, p_{start}, p_{goal})

输入：一组互不相交的多边形障碍物，位于自由空间中的两个点 p_{start} 和 p_{goal}

输出：由 p_{start} 通往 p_{goal}、无冲突的一条最短路径

1. $\mathcal{G}_{vis} \leftarrow$ VISIBILITYGRAPH($S \cup \{p_{start}, p_{goal}\}$)
2. 对于 $\mathcal{G}_{vis}$ 中的每一条弧(v, w)，
   根据线段$\overline{vw}$的长度，为其指定一个权值

3. 采用 Dijkstra 算法， 在 $\mathcal{G}_{vis}$ 中计算出一条由 p_{start} 通往 p_{goal} 的最短路径

下一节将说明，如何在 $O(n^2\log n)$时间内构造一张可见性图（其中 n 为各障碍物所含边的总条数）。显然，$\mathcal{G}_{vis}$ 最多含有$\binom{n+2}{2}$条弧。因此，该算法第 2 行需要 $O(n^2)$时间。若图中共有 k 条弧，且其权值均非负，则 Dijkstra 算法可以在 $O(n\log n + k)$时间内计算出两个节点之间的最短路径。在这里，$k = O(n^2)$，由此可得出结论：SHORTESTPATH 的总体运行时间为 $O(n^2\log n)$。这可以总结为如下定理：

〖定理 15.3〗

穿行于一组互不相交的多边形障碍物之间、从 p_{start} 通往 p_{goal} 的任何一条最短路径，都可以在 $O(n^2\log n)$ 时间内构造出来，其中 n 为各障碍物所含边的总数目。

15.2 构造可见性图

设 S 为平面上一组互不相交的多边形障碍物，其中边的总数为 n。（在上一节中，算法 SHORTESTPATH 需要构造出集合 S^*的可见性图，这里的 S^*加入了起点和终点。尽管出现了这两个孤立顶点（isolated vertex），但不会引起任何问题，因此本节将不会显式地去专门处理它们。）为构造出 S 的可见性图，必须找出所有相互可见的顶点对。这就是说，对于每一对顶点，都需要进行测试，以判断联接它们的线段是否与某个障碍物相交。如果直截了当地这样做，每一次测试都需要 $O(n)$时间，而总的运行时间将高达 $O(n^3)$。我们很快就会看到，完全可以更加有效地进行这种测试——为此，不能按照随意的次序来考虑各顶点对，而应该每次只将注意力集中在某一个顶点，并检查其它各顶点是否与之可见。这可以描述为如下算法：

```
算法 VISIBILITYGRAPH(S)
输入：一组互不相交的多边形障碍物 S
输出：可见性图 Gvis(S)
1.   对图 G = (V, E)做初始化，
        使得集合 V 包括 S 中所有多边形的顶点，E = ∅
2.   for (每一顶点 v ∈ V)
3.      do W ← VisibilityVertices(v, S)
4.         对于每一顶点 w ∈ W，将弧(v, w)添加到 E 中
5.   return G
```

其中子过程VISIBILITYVERTICES的输入包括一组多边形障碍物S，以及平面上的一个点p；在这里，

p是S中的一个顶点，但这一点并不是必需的。如 图 15-7 所示，该子过程将在各障碍物的顶点中，找出并返回与p可见的所有顶点。

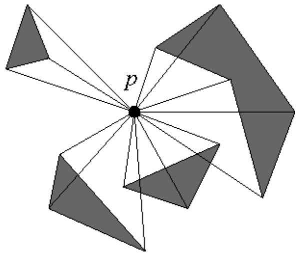


图15-7 测试S中每一顶点到p的可见性

若仅需测试某一特定顶点 w 是否与 p 可见，则并没有多大的改进余地——只能将线段$\overline{pw}$与各障碍物逐一进行测试。然而，要是需要对 S 中的所有顶点进行测试，则具体做法的确有些讲究——对一个顶点做过测试之后，可以利用已获得的信息加速对此后各顶点的测试。让我们来考虑所有线段$\overline{pw}$所构成的集合。应该按照何种次序来处理它们，才能利用由前面顶点给出的信息，来处理后面的顶点呢？符合逻辑的一种方案是，按照围绕 p 的环形次序。这样，所需要做的就是按照这一环形次序来处理各个顶点，并动态地维护某些信息，以帮助我们对下一待处理的顶点做出判断。

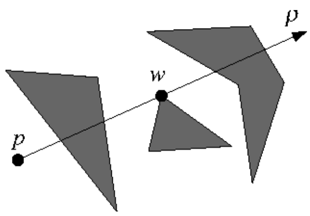


图15-8 w与p不可见

顶点w与p可见的条件是，线段$\overline{pw}$不与任何障碍物相交于内部。考察从p发出、穿过w的那条射线ρ。如 图 15-8 所示，若w与p不可见，则在到达w之前，ρ必然先碰到某个障碍物的边。为了对此进行测试，我们可以在与ρ相交的所有障碍物边中进行二分查找。这样，就可以判断（从p的位置来看）w是否处于其中某条边的后方。（若p本身就是障碍物的一个顶点，则还有一种w不可见的情况——p和w都是同一障碍物的顶点，而且$\overline{pw}$穿过该障碍物的内部。只要检查与w相关联的各边，就可以判断在到达w之前，ρ是否进入了障碍物的内部，从而确定是否属于这种情况。当$\overline{pw}$包含与w关联的某条边时，将会出现退化情况，对此我们姑且不予考虑。）

因此，在按照围绕p的环形次序处理各顶点的过程中，要将与ρ相交的各边组织成一棵二分查找树T。（正如我们稍后将会看到的，被ρ包含的边都不需要存放到T中。）T的各匹叶子，依次存放了各条相交边：最左侧的叶子存放的是与ρ相交的第一条边；第二匹叶子存放了下一条相交边；依此类推。

为了能够引导查找，树中各内部节点也存放有边。更准确地说，在内部节点v处存放的，是其左子树中最右侧（叶子处存放）的边e_v——因此按照由ρ确定的次序，其右子树中的所有边都大于e_v，左子树中的所有边都小于或等于e_v。图 15-9 给出了这样一个例子。

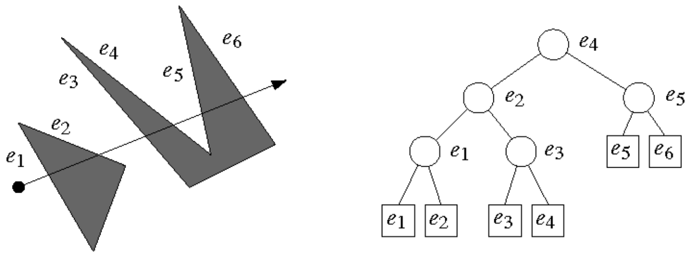


图15-9 对相交边的查找树

按照环形次序有效地处理各顶点，就是围绕 p 旋转射线ρ。因此，这里采用的办法类似于我们曾经在其它场合多次使用过的平面扫描模式；差别在于，这里使用的不再是一条自上而下扫过平面的水平线，而是一条旋转的射线。

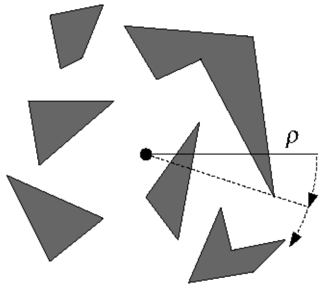


图15-10 旋转式平面扫描

如 图 15-10 所示，对于这种旋转式平面扫描（rotational plane sweep）而言，所谓的状态（status）就是与（当前）ρ相交各障碍物的有序序列。它可以通过$\mathcal{T}$来维护。扫描过程中的事件，就是S中的各顶点。在处理顶点w时，我们必须对状态结构$\mathcal{T}$进行查找，以判断w是否与p可见；然后，还要对$\mathcal{T}$做相应的更新，插入或删除与w关联的某些障碍物边。

我们的旋转式平面扫描过程，可总结为算法 VISIBLEVERTICES。扫描线ρ的起始方向为正 x 方向；然后，沿顺时针方向旋转。因此，该算法首先要根据 p 与各顶点的联线和 x-轴正方向的顺时针夹角，对所有顶点排序。要是有两个甚至更多顶点的夹角相等，又该如何呢？为了能够正确判断出某个顶点 w 的可见性，需要知道$\overline{pw}$是否与某个障碍物相交于内部。因此，一种显而易见的策略就是：在处理 w 之前，需要处理完落在$\overline{pw}$内部的所有顶点。换而言之，对（与 x-轴）夹角相等的多个顶点，必须以它们到 p 距离的递增顺序进行处理。这样，就得到了形式如下的一个算法：

算法 VISIBLEVERTICES(p, S)

输入：一组多边形障碍物 S
　　不在任何障碍物内部的一个点 p

输出：与 p 可见的所有障碍物顶点

1. 根据 p 与障碍物各顶点所确定的射线与 x-轴正向所成的顺时针夹角，
　　对所有顶点排序
　要是出现夹角相等的情况，距离 p 更近的顶点排在前面
　令排序后的顶点列表为：$w_1, \ldots, w_n$

2. 令ρ为从 p 出发、平行于 x-轴正向的那条射线
　找出与ρ真相交的所有障碍物边
　将它们按照与ρ相交的次序，存放到一棵二分查找树 T 中

3. W ← 0

4. **for** i ← 1 **to** n

5. 　**do if** Visible(w_i) **then** 将 w_i 加入到 W 中

6. 　　在与 w_i 关联的各障碍物边中，找出落在射线$\overline{pw_i}$顺时针一侧的所有边
　　将这些边插入到 T 中

7. 　　在与 w_i 关联的各障碍物边中，找出落在射线$\overline{pw_i}$逆时针一侧的所有边
　　将这些边从 T 中删除

8. **return** W

子程序 Visible 必须能够判断顶点 w_i 是否可见。通常，这只需要对 T 进行查找，检查与 p 最靠近的那条边（即在最左侧叶子中存放的边）是否与$\overline{pw_i}$相交。但要是$\overline{pw_i}$上还有其它顶点，我们就要格外小心。在这种情况下，究竟 w_i 是不是可见的呢？这要是具体情况而定。

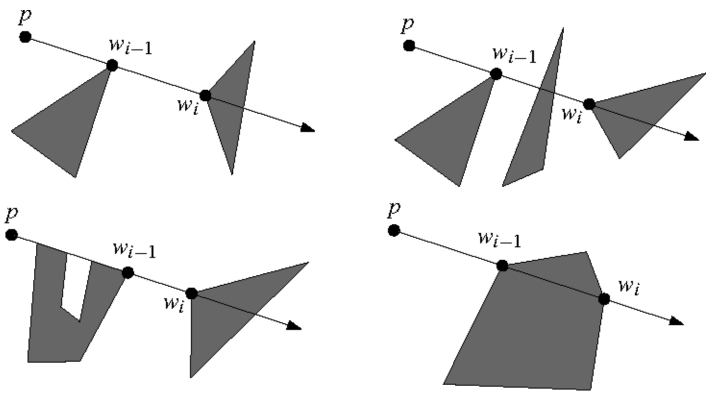


图15-11　ρ同时穿过多个顶点的几种情况。
在所有这些情况中，w_{i-1}都是可见的。左侧的两种情况下，w_i也是可见的；在右侧的两种情况下，w_i是不可见的

图 15-11 给出了可能的几种情况。在与这些顶点相关联的各障碍物中，$\overline{pw_i}$可能会与某些相交于其内部，也可能不会。看起来，似乎只能通过逐一检查那些有一个顶点落在$\overline{pw_i}$上的边，才能判断到底w_i

是否可见。幸运的是，在此前处理落在$\overline{pw_i}$上的各个顶点时，实际上已经检查过相应的边了。因此，可以按照如下方法来判断w_i的可见性。若w_{i-1}不可见，则w_i必然也不可见。即使w_{i-1}可见，w_i仍然有两种可能是不可见的——要么w_{i-1}和w_i都属于同一障碍物，而且线段$\overline{w_{i-1}w_i}$完全落在该障碍物的内部；要么线段$\overline{w_{i-1}w_i}$与$\mathcal{T}$中的某条边相交。（若是后一种情况，则这条边夹在w_{i-1}与w_i之间，因此它必然与$\overline{w_{i-1}w_i}$真相交。）这种测试是正确的——因为$\overline{pw_i} = \overline{pw_{i-1}} \cup \overline{w_{i-1}w_i}$。（若$i = 1$，则p和$w_i$之间将不会有任何顶点，于是此时只需检查线段$\overline{pw_i}$。）我们可以得出如下子程序：

算法 VISIBLE (w_i)

1. **if** ($\overline{pw_i}$与 w_i 所属的障碍物相交于内部，就局部而言就在 w_i 处)
2. **then return false**
3. **else if** ($i = 1$) **or** (w_{i-1} 不在线段$\overline{pw_i}$上)
4. **then** 找出 $\mathcal{T}$ 中最左侧的叶子，取出其中存放的边 e
5. **if** (e 存在，而且$\overline{pw_i}$与 e 相交)
   then return false
   else return true
6. **else if** (w_{i-1} 不可见)
   then return false
7. **else** 在 $\mathcal{T}$ 中查找与$\overline{w_{i-1}w_i}$相交的一条边 e
8. **if** (e 存在)
   then return false
   else return true

至此，我们完成了对算法 VISIBLEVERTICES 的描述。利用该算法，可以计算出与给定点 p 可见的所有顶点。

VISIBLEVERTICES 的运行时间如何？在第 4 行之前，计算时间主要花费于按照围绕 p 的环形次序，对各顶点进行排序，这部分时间为 $O(n\log n)$。接下来的每一轮循环，都只涉及对二分查找树 $\mathcal{T}$ 的常数次操作，这需要 $O(\log n)$时间；此外，还有常数次的几何测试，每次测试都可以在常数时间内完成。因此，每轮循环需要 $O(\log n)$时间——于是，总体运行时间就是 $O(n\log n)$。

你应该记得，为了构造出整个可见性图，需要对 S 中的 n 个顶点逐一使用 VISIBLEVERTICES 算法。这样，我们就得到了如下定理：

〖定理 15.4〗

任意给定一组互不相交的多边形障碍物 S，其可见性图可以在 $O(n^2\log n)$时间内构造出来，其中 n 为 S 中边的总数。

15.3 平移运动多边形机器人的最短路径

在第 13 章中我们已经看到，通过构造出自由C-空间（free configuration space），可以将关于平移运动、凸多边形（convex polygon）形状机器人R的运动规划问题，化简为点机器人的情况。这种化简的方法，首先需要计算出$-R$（即R的对称镜像）与每个障碍物的Minkowski和，然后再记下所得出的C-空间障碍物的并集。由此得出的，是一组互不相交的多边形，它们合起来就构成了禁止C-空间（forbidden configuration space）。

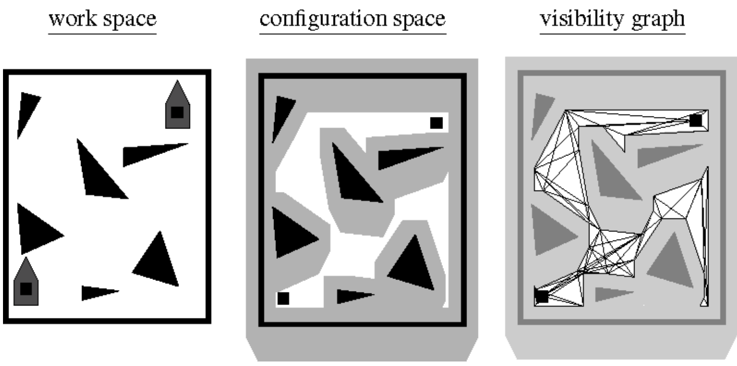


图15-12　多边形机器人的最短路径规划：工作空间（左），C-空间（中），可见性图（右）

这样，就可以套用针对点机器人的方法来计算最短路径——先将起点和终点在 C-空间中对应的两个点，扩充到多边形集合中；然后构造这些多边形对应的可见性图，其中每一条弧的权值，分别被赋为对应的可见边的欧氏长度；最后，运用 Dijkstra 算法，在可见性图中找出一条最短路径。

按照这一方法，运行时间将是多少呢？首先，根据〖引理 13.13〗，可以在$O(n\log^2 n)$时间内构造出禁止空间；此外，根据〖定理 13.12〗，禁止空间的复杂度为$O(n)$。因此，由前一节的结论可知，禁止空间的可见性图可以在$O(n^2\log n)$时间内构造出来。

由此可以总结出下述定理：

〖定理 15.5〗

设机器人 R 的形状是凸的，其复杂度为常数，可以在一组多边形障碍物之间做平移式运动。对于任何给定的起始和目标位置，我们都能够在 $O(n^2\mathbf{log}n)$时间内，为 R 规划出一条不发生碰撞的最短路径。其中 n 为所有障碍物所包含边的总条数。

15.4 注释及评论

对于在带权图中规划最短路径这一问题，人们已经做过详尽的研究。大多数有关图算法或者算法及数据结构的书籍，都会介绍Dijkstra算法以及此类的其它解法。第 15.1 节曾经指出，Dijkstra算法的运行时间为$O(n\mathbf{log}n + k)$。为使时间上界（upper bound）降至这一数量，必须借助Fibonacci堆（Fibonacci heap）来实现该算法。就此处的具体应用而言，$O((n+k)\mathbf{log}n)$的算法就足够了——因为无论如何，算法的其余部分都要消耗这样多的时间。

对于最短路径问题的几何版本，人们也已经有了相当多的关注。这里所介绍的算法，来自Lee[247]。人们还提出过更加高效、基于排列（arrangement）的算法，它们的运行时间为$O(n^2)$。

无论是采用哪种算法来规划最短路径，只要该算法需要首先构造一幅完整的可见性图，那么就最坏情况而言，它就注定需要运行至少平方量级的时间——因为，可见性图中包含的边数可能高达平方量级。在很长一段时间内，人们一直没有找到最坏情况运行时间低于平方量级的算法。平方量级的界限，是由Mitchell[281]首先打破的。他证明，可以在$O(n^{5/3+\varepsilon})$时间内，为点机器人规划出最短路径。后来，他 [282]又将自己算法的运行时间改进为$O(n^{3/2+\varepsilon})$。然而与此同时，Hershberger和Suri[210][212]成功地构造出了一个时间复杂度为$O(n\mathbf{log}n)$的最优算法。

若机器人的自由空间是一个不含孔洞的多边形，则在这种特殊情况下，可以在线性时间内规划出一条最短路径——为此，需要将Chazelle[94]的线性时间三角剖分算法，与Guibas等人 [195]的最短路径方法结合起来。

欧氏最短路径问题的三维版本更难求解。这是因为，没有什么方法可以轻易地离散化（discretize）该问题——沿最短路径上的转折点（inflection point），并不会限制于有限的一组候选点；实际上，它们可以是障碍物边上的任何点。Canny[80]曾证明，在一组三维多面体障碍物之间规划出联接两个点的最短路径，是NP-难的（NP-hard）问题。Reif和Storer[327]将该问题归约为实数理论中的一个判定问题，从而得到了求解该问题的一个单指数算法（single-exponential algorithm）。还有几篇论文，给出了在多项式时间内得到近似最短路径的方法。例如，先在障碍物边上增加点，并以这些点为节点构造出一张图，然后在这张图中进行查找 [13][126][125][260][316]。

本章主要讨论了欧氏度量（Euclidean metric）。也有很多论文讨论了不同度量下的最短路径。其中可变的因素太多，在此只能提及其中的几种，而且针对每一因素，也只能提供少量的参考文献。在被研究过的各种度量中，一种有趣的度量被称作链环度量（link metric）。在这种度量下，一条多边形路径的长度被定义为该路径上所含的链环数目 [367][284][20][122]。另一种被深入研究过的情况，是所谓的直角路径（rectilinear path）。例如在VLSI设计中，这种路径将扮演重要的角色。针对直角路径的问题，Lee等人 [253]曾经做过综述。关于直角路径，人们研究过的一种有趣的度量就是组合度量（combined metric）[56]。所谓组合度量，就是欧氏度量和链环度量的线性组合。最后，有的论文还研究了子区域划分（subdivision）中的路径，其中每个子区域都被赋予一定的权重。路径穿越某个子区域的代价，等于穿越的欧氏长度乘以该子区域的权重。通过给子区域赋予无限大的权重，可以模拟障碍物 [113][283]。

对于平移式机器人而言，有多种显而易见的度量（比如，你马上就会想起欧氏度量）。然而，要是机器人不仅可以平移，同时还能旋转，那么要想为最短路径给出一个好的定义，并非易事。对于（可以表示为）线段的机器人，人们已经获得了一些结果 [24][114][218]。

利用可见性图进行运动规划的想法，始于Nilsson[295]。这里介绍的用$O(n^2 \log n)$时间构造可见性图的算法，来自Lee[247]。此外，已经找到了若干更快速的算法 [23][383]，其中包括Ghosh和Mount[190]提出的最优算法，该算法的运行时间为$O(n \log n + k)$，其中k为可见性图中弧的总条数。

若是在一组凸的多边形障碍物之间为点机器人规划一条最短路径，则并不需要用到所有的可见边。实际上，只要所有定义了公切线的可见边就足矣。Rohnert[329]给出了一个算法，可在$O(n + c^2 \log n)$时间内构造出这种精简的可见性图，其中c为障碍物的数目，n为所有障碍物所含边的总条数。

Vegter和Pocchiola[319][320][376]提出的可见性复形（visibility complex）结构，复杂度与可见性图相同，却包含了更多的信息。可以对平面上任意一组凸的（形状不见得是多边形的）物体定义这种结构，借助该结构，可以求解最短路径问题和光线发射（ray shooting）问题。这种结构可以在$O(n \log n + k)$时间内构造出来。

15.5 习题

习题 15.1 设 S 为平面上一组互不相交的简单多边形，其中包含边的总条数为 n。试证明：对于任何起点和终点，组成最短路径的线段条数不会超过 $O(n)$。试给出一个$\Theta(n)$例子。

习题 15.2 对每条障碍物边，算法 VISIBILITYGRAPH 都要调用一次算法 VISIBLEVERTICES。VISIBLEVERTICES要将所有顶点围绕其输入点排序。这就意味着，对每个障碍物顶点都

要做一次环形排序，总共要做n次。本章中，分别做一次排序的时间是$O(n\log n)$，这样就使得全部的排序需要$O(n^2\log n)$时间。试证明：通过对偶变换（参见第 8 章），可以改进为$O(n^2)$。这样做能够改善VISIBILITYGRAPH算法的总体时间复杂度吗?

习题 15.3 规划最短路径的算法，可以从多边形扩展到其它形状的物体。设 S 为一组共 n 个互不相交的圆盘状（disc-shaped）障碍物，各障碍物的半径不一定相等。

a. 试证明：在这种环境里，相互不能直接可见的两个点之间的最短路径，由这些圆盘的部分边界、这些圆盘之间的公切线以及起点和终点到各圆盘的切线联接而成。

b. 将可见性图的概念应用到这种环境中。

c. 应用最短路径算法，在任意两点之间，规划出一条穿行于 S 中各圆盘之间的最短路径。

习题 15.4 对于平面上（互不相交的）n 个三角形（障碍物），联接于两个固定点之间的最短路径最多会有多少条?

习题 15.5 设 S 为一组互不相交的多边形，给定起点 p_{start}。我们希望通过对集合 S（以及 p_{start}）的预处理，使得对任意终点，都能够有效地规划出从 p_{start} 到该终点的最短路径。试说明，如何才能在 $O(n^2\log n)$时间内完成预处理，使得对于任意终点 p_{goal}，都可以在$O(n\log n)$时间内在 p_{start} 和 p_{goal} 之间规划出一条最短路径。

习题 15.6 试设计一个算法，在一个简单多边形内的任意两点之间，规划一条最短路径。算法的运行时间不得超过平方量级。

习题 15.7 若所有障碍物都是凸多边形，则最短路径算法还可以改进——这种情况下，如 图 15-13 所示，只需考虑公切线，而不是所有的可见边。

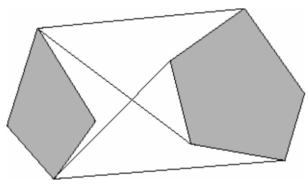


图15-13 凸障碍物

a. 试证明：此时可能组成最短路径的可见边，只可能是各多边形之间的公切线。

b. 试给出一个快速算法，找出互不相交的两个凸多边形之间的公切线。

c. 试给出一个算法，在一组凸多边形之间，找出所有是可见边的公切线。

习题 15.8* 要是你熟悉齐次坐标（homogeneous coordinate），就会发现一个有趣的事实：本章所采用的旋转式平面扫描，可以转换为通常的（使用一条水平直线、平移扫过整个平面的）平面扫描。试说明：这种转换，就是通过投影变换（projection transformation），将（旋转式平面扫描）的中心变换到无穷远点。

16

单纯形区域查找：再论截窗

在第 2 章中我们曾经看到，地理信息系统常常需要将一幅地图存储为若干独立的图层。每一图层分别代表该地图的某一主题（theme）——亦即，专门的一类（地理）特征，比如公路、城市等等。将地图中的信息化分为不同的图层，可以使用户能够将注意力集中于某一特定的特征。即使是在描述同一类信息的一幅专题图中，人们也不见得会对其中所有的特征感兴趣，而往往只是对某一范围内的局部感兴趣。第 10 章就曾给出过这样一个例子：根据整个美国的一幅路线图，抽取出很小的某一区域之内的部分。在那一章所讨论的这类问题中，待查询区域——也称为截窗（window）——是矩形的。但实际上，我们很容易就可以举出待查询区域为其它形状的例子。

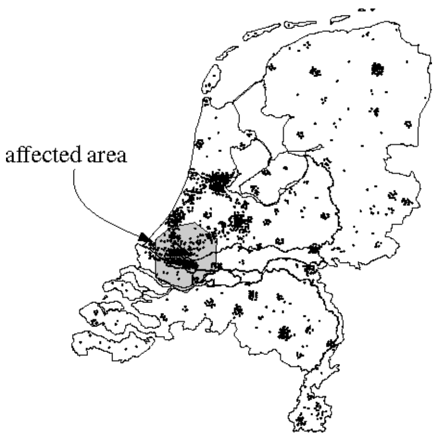


图16-1 荷兰的人口分布密度

假设我们拿到关于人口分布密度的一幅专题图。图中为表示密度，每 5,000 个人都用一个点来代表，如 图 16-1 所示。现在计划在某处修建飞机场，如果需要事先估计出该飞机场的影响，就需要掌握居住在受飞机场影响区域之内的人口数量。用几何的语言可表述为：给定平面上一组点，需要统计出落在某一特定待查询区域（query region，比如飞机噪声超过某一级别的区域）内的点数。

第 5 章讨论过数据库的查询问题：给定与坐标轴平行的某一待查询矩形，报告出落在其中的所有点。为此曾专门设计了一种数据结构。然而，受到飞机场影响的区域取决于主风向，不大可能是矩形的，故第 5 章中的数据结构在这里没有多大用处。我们需要设计出一种能够支持更一般性待查询区域的数据结构。

16.1 划分树

给定平面上一组点，我们希望统计落在某一特定待查询区域中的点数。（从现在起，我们将按照习惯，将“统计……点数”理解为“报告出……点的数目”，而不是逐一枚举出各点。）假定待查询区域是一个简单多边形（simple polygon）——否则，总是能够用简单多边形来近似它。为简化查询算法，首先要对待查询区域做三角剖分（亦即，将它分解为若干三角形）。第 3 章介绍过三角剖分的具体算法。一旦完成待查询区域的三角剖分，就可以针对其中每一个三角形进行查询。分别落在各三角形之内的点合在一起，就是落在整个区域内的那些点。若是需要统计点的数目，还得更加小心，不要对同时落在两个三角形之间公共边上的点重复计数——好在这并不困难。

这样，就将问题转化为所谓的三角形区域查找问题（triangular range searching problem）：给定由平面上n个点组成的一个集合S，统计出S中有多少点落在某一特定待查询三角形t之内。首先来讨论该问题稍做简化之后的一个版本：如 图 16-2 所示，待查询三角形退化为一张半平面。

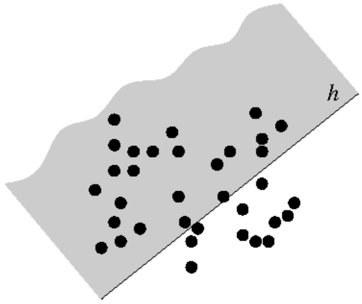


图16-2 半平面区域查询问题

为支持这类半平面区域查询（half-plane range query），需要什么样的数据结构呢？先来看一维的情况，然后再循序渐进。该问题的一维版本中，给定的输入是分布在实轴上的一组共n个点，目标是统计出其中落在某一待查询射线（query half-line）上的点数（亦即落在某一待查询点某一侧的点数 ）。只要使用一棵平衡二分查找树，将其中各子树内所含的点数记录到相应的节点之中，就可以在$O(\log n)$时间内回答每次查询。如何将这一方法推广到二维情况呢？为回答这一问题，必须首先从几何的角度来理解平衡二分查找树。树中每一节点都存有一个关键码——也就是点的坐标——根据这个关键码，可以点集一分为二，分别存放到左、右子树中。相应地，也可以将这个关键码的作用，理解为将整个实轴切成两段。按照这种理解，树中的每个节点都对应于实轴上的某一区间——根节点对应于整个实轴，根节点的两个孩子分别对应于一条射线，如此切分下去。对于任一待查询射线，在每个节点的两个孩子中，必有一个要么完全包含在射线之中，要么完全落在射线的外面。于是，这个（孩子所对应）区间内的所有点，要么全部落在射线之内，要么没有一个落在其内。因此，只需对该节点的另一个孩子递归地查询下去。这一过程如 图 16-3 所示。

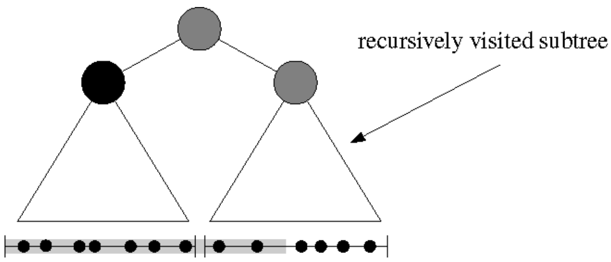


图16-3 利用二叉树来支持射线区域查询（half-line range query）

该图中，在树的下方用黑点画出了所有的点；两棵子树在实轴上各自对应的区间，也分别做了标记加以区分。这里的待查询射线为灰色的那段区域。以黑色节点为根节点的子树所对应的区域，完全落在待查询射线之内。因此，只需对右子树进行递归查询。

为将上述方法推广到二维，你也许会指望能够将平面划分为两个子区域，使得对于任一待查询半平面，都有一个子区域要么完全包含在半平面之内，要么完全落在其外。遗憾的是，这样的划分并不存在，因此需要做进一步的推广——既然两个子区域不够，就必须划分出更多个子区域。这样的划分应该使得对于任一待查询半平面，都只需对其中的少数几个子区域做递归搜索。

我们所需要的这种划分，可以形式化地描述如下。对于平面上由任意 n 个点组成的集合 S，所谓 S 的一个单纯形划分（simplicial partition）就是一个集合 $\Psi(S) := \{(S_1, t_1), \ldots, (S_r, t_r)\}$，其中的 S_i 都是 S 的子集，它们互不相交，且它们的并集为 S，而 t_i 分别为包含 S_i 的一个三角形。每一个子集 S_i 被称为一个类（class）。各三角形不必互不相交，故 S 中的某一点可能同时落在多个三角形中。不过，这样的一个点总是属于唯一的某个类。三角形的总数称为 Ψ(S)的规模，记作 r。

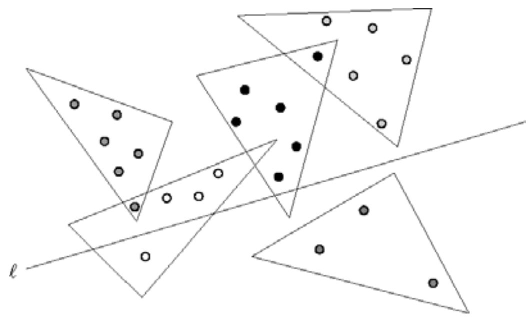


图16-4 一个好的单纯形划分

图 16-4 就是单纯形划分的一个例子，其规模为 5。该图中，不同的灰度表示不同的类。我们称一条线l穿越了一个三角形t_i，如果l与t_i的内部相交。如果点集S不是处于一般性位置的（in general position），那么为了构造单纯形划分，不仅要使用三角形，有时也需要使用（相对开的）线段。我们称一条直线穿越了这样的一条线段，如果它与该线段的相对内部相交，却又不是完全包含该线段。对于任一直线l，所谓l相对于Ψ(S)的穿越数（crossing number），就是在Ψ(S)中与l相交的三角形数目。因此对图 16-4 中那条直线而言，穿越数为 2。而Ψ(S)自己的穿越数，则等于所有直线可能对应的最大穿越数。在 图 16-4 中，可以找出穿越四个三角形的一条直线，但是任何直线都不可能同时穿越这五个三角形。最后，我们称一个单纯形划分是好的（fine），如果对于任何 $1 \le i \le r$，都有$|S_i| \le 2n/r$。换而言之，在好的单纯形划分中，任一类点的数目都不会超过平均点数的两倍。

在对划分做过形式化定义之后，接下来我们要看看，如何才能利用这种划分来回答半平面区域查找。设h为一张待查询半平面。如果划分出来的某个三角形t_i没有被h的边界线穿越，那么类S_i要么整体包含于h之中，要么整体处于h之外。这样一来，我们就只需要进一步考虑h的边界线所穿越的那些t_i，并对其相应的S_i进行递归查询。以 图 16-4 为例，如果针对l^+（即位于l上方的那张半平面）进行查询，就只需要对全部五个三角形中的两个进行递归查询。因此，这一回答查询过程的效率，将取决于单纯形划分的穿越数——穿越数越小，查询时间也就越短。下面的定理指出：总是可以构造出一个穿越数为$O(\sqrt{r})$的单纯形划分。稍后我们将看到，这对于查询时间意味着什么。

〖定理 16.1〗

由平面上任意 n 个点构成的任一集合 S，对于任意 $1 \le r \le n$，都必然存在一个规模为 r、穿越数为 $O(\sqrt{r})$ 的单纯形划分。此外，对于任何$\varepsilon > 0$，都可以在 $O(n^{1+\varepsilon})$时间内构造出这样的一个单纯形划分。

该定理声称构造时间为 $O(n^{1.1})$、$O(n^{1.01})$或者任一更接近于 1 的指数，看起来有点奇怪。但事实上，无论ε多么小，只要它是一个正的常数，就总是能够得到该定理所提到的时间复杂度上界（upper bound）。不过，这个定理并没有提出一个（诸如 $O(n)$或 $O(n\log n)$的）更好上界。

在第 16.4 中你可以找到一个文献索引，该文献给出了上述定理的证明。这里直接假定这一定理是成立的，而把注意力集中于如何利用该定理来设计出一种能够有效解决半平面区域查找问题的数据结构。我们将要导出的数据结构称为划分树（partition tree）。或许，你已经猜测出一棵划分树的样子：这棵树的根节点有r个孩子；分别以每个孩子为根节点，可以递归地定义出对应于单纯形划分中同一类点的一棵子树。各孩子之间没有特定的次序；这一点恰好是无所谓的。

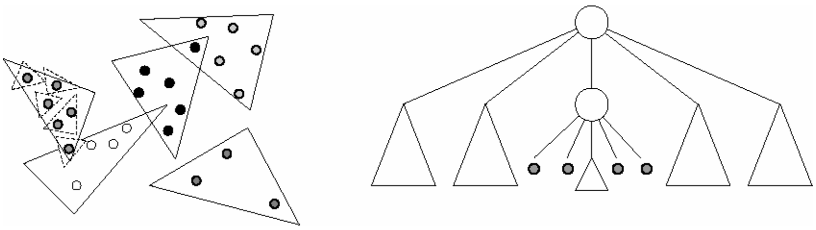


图16-5 一个单纯形划分及其对应的划分树

在 图 16-5 中，给出了一个单纯形划分及与之对应的划分树。请注意根节点居于中间的那个孩子。如果我们对它所对应的那一类进行递归划分，将得到若干个（更小的）三角形——也就是 图 16-5 中用虚线指示的那几个三角形。于是，这一类又被进一步划分为 5 个“子类”（subclass），分别存放在根节点中间那个孩子下面的 5 棵子树中。视具体应用的要求，可能还要记录一些关于子类的附加信息。这样，一棵划分树的基本结构可以定义如下：

- 若 S 只包含一个点 p，则划分树由单独的一匹叶子组成，点 p 被显式地存放在这匹叶子中。集合 S 就是该叶子对应的正则子集。
- 否则，该结构就是分支度（branching degree）为 r 的一棵树 $\mathcal{T}$，其中 r 为一个足够大的常数（稍后将介绍 r 的取法）。集合 S 存在一个规模为 r 的、好的单纯形划分，而树 $\mathcal{T}$ 根节点的各个孩子，就与该划分中的各三角形一一对应。在该划分中，与孩子节点 v 对应的那个三角形记作 t(v)。而 S 中对应的那一类，则被称为 v 的正则子集，记作 S(v)。以孩子节点 v 为根节点，可以针对集合 S(v)递归地定义出一棵划分树。
- 在每个孩子节点 v 处，都存放有三角形 t(v)。此外，还要存放有关子集 S(v)的某些信息——就半平面区域计数（half-plane range counting）问题而言，这些信息就是 S(v)的基数；若是

其它的应用问题，则需要存放其它类型的信息。

接下来，针对“统计 S 中落在指定待查询半平面 h 内的点数”这一问题，我们来给出一个查询算法。这个算法将返回来自划分树 $\mathcal{T}$ 的一组节点，称作拣出节点（selected node）；由这些节点构成的集合记作 Υ。这组节点具有如下性质：它们各自所对应正则子集的无交并集，将给出 S 中落在 h 内的所有点。换而言之，集合 Υ 中各节点所对应的正则子集互不相交，而且

$$S \cap h = \bigcup_{v \in \Upsilon} S(v)$$

实际上，这些拣出节点恰好就是全部具有如下性质的节点 v：$t(v) \subset h$（或者，若 v 是一匹叶子，则存放在 v 中的那个点落在 h 中），而且 v 的任何一个祖先μ都不满足 $t(\mu) \subset h$。只要将拣出的正则子集的基数累加起来，也就得到了被 h 包含的点数。

```
算法 SELECTINHALFPLANE(h, 𝒯)
输入：待查询半平面 h，及其对应的一棵划分树或子树 𝒯
输出：一组正则节点，它们给出了树 𝒯 中落在 h 内的所有点
1.   ϒ ← ∅
2.   if (𝒯 是由一匹叶子μ构成的)
3.      then if (存放于μ处的点落在 h 内)  then ϒ ← {μ}
4.      else for (𝒯 的根节点的每一个孩子 v)
5.               do if (t(v) ⊂ h)
6.                     then ϒ ← ϒ ∪ {v}
7.                     else if (t(v) ∩ h ≠ ∅)
8.                             then ϒ ← ϒ ∪ SELECTINHALFPLANE(h, 𝒯_v)
9.   return ϒ
```

以上查询算法的过程如 图 16-6 所示。根节点的孩子中被拣出的用黑色表示。被递归访问到的那些孩子，则用灰色表示（包括根节点本身，因为它也被访问到了）。正如此前所言，为了回答每一次半平面区域计数查询，只要调用SELECTINHALFPLANE，并将所有拣出节点的基数累加起来即可。请注意，每个节点都保存了其对应正则子集的基数。在实际应用中，也许甚至不用动态维护集合ϒ，而只需一个计数器即可——每拣出一个节点，就将与之对应的正则子集的基数累计到计数器中。

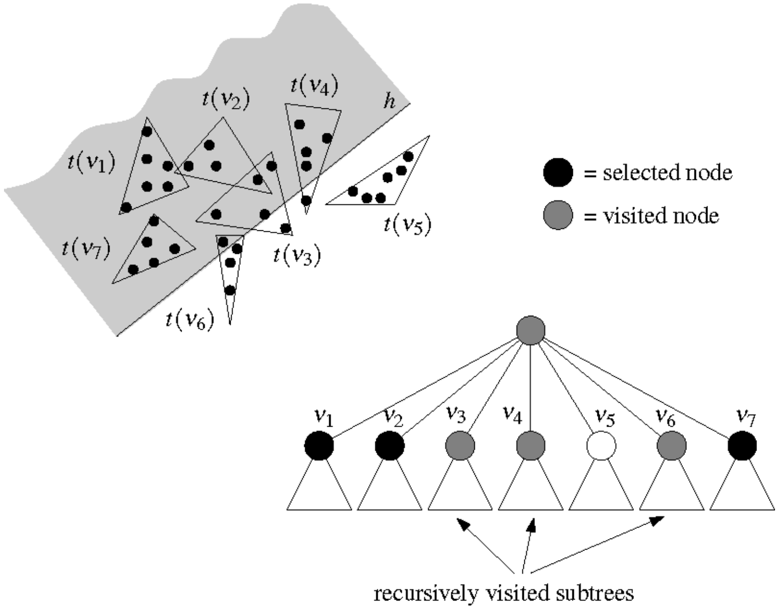


图16-6 借助划分树来回答半平面区域查找

以上介绍了划分树——一种支持半平面区域计数查询的数据结构——以及对应的查询算法。接下来，需要对该结构做一分析。首先是它占用的存储量。

〖引理 16.2〗

设 S 为平面上的一组共 n 个点。对应于 S 的一棵划分树只需要占用 $O(n)$的存储空间。

〖证明〗

对于任意 n 个点，可能与之的所有划分树中所包含节点数目的最大值记作 M(n)；将正则子集 S(v)的基数记作 n_v。则 M(n)满足如下递推式：

$$M(n) \leq \begin{cases} 1 & 若 n = 1 \\ 1 + \sum_v M(n_v) & 若 n > 1 \end{cases}$$

其中的和式，取遍该树根节点的所有孩子。既然单纯形划分中的任意两类之间都没有公共点，故有$\sum_v n_v = n$。另外，所有的 v 都满足 $n_v \leq 2n/r$。因此，对于任何常数 $r > 2$，上面的递推式的解都是 $M(n) = O(n)$。

树中每一节点都只占用 $O(r)$空间。由于 r 是一个常数，故本引理得证。 ❑

线性规模的存储空间，已经没有改进余地了。那么，查询时间呢？要回答这个问题，确切的 r 值将很重要。为了完成一次查询，需要对不超过 $c\sqrt{r}$棵子树进行递归查询，其中的 c 是一个与 r 和 n 无关的常数。实际上，该常数会对查询时间上界中 n 的具体指数有所影响。为了降低这种影响，需

要取足够大的 r——正如我们稍后将要看到的，这样才能将查询时间降低到非常接近于 $O(\sqrt{n})$。

〖引理 16.3〗

设 S 为平面上任意 n 个点。对于任何 $\varepsilon > 0$，都存在 S 的一棵划分树，使得对任一平面 h，都可从该树中拣出 $O(n^{1/2+\varepsilon})$个节点，它们所对应正则子集的无交并集恰好给出了 S 中落在 h 内的所有点。为拣出这些节点，需花费 $O(n^{1/2+\varepsilon})$时间。这样，每次半平面区域计数查询都可以在 $O(n^{1/2+\varepsilon})$时间内完成。

〖证明〗

任意给定$\varepsilon > 0$。根据〖定理 16.1〗，必存在一个常数c，使得对于任意参数r，都可以构造出一个规模为r、穿越数不超过$c\sqrt{r}$的单纯形划分。我们取$r := \lceil 2(c\sqrt{2})^{1/\varepsilon} \rceil$，然后以r为规模构造出一个单纯形划分，同时构造出与之对应的划分树。在所有对应于n个点的划分树中，单次查询所需的最长查询时间记作Q(n)。任取一张待查询半平面h，将正则子集S(v)的基数记作n_v。于是，Q(n)满足如下递推关系：

$$Q(n) \leq \begin{cases} 1 & \text{若}n = 1 \\ r + \sum_{v \in C(h)} Q(n_v) & \text{若}n > 1 \end{cases}$$

其中的和式取遍集合 C(h)——它由该树根节点满足如下性质的那些孩子 v 组成：t(v)与 h 的边界相交。既然该数据结构背后的单纯形划分的穿越数为 $c\sqrt{r}$，集合 C(h)所含节点的数目也就不会超过 $c\sqrt{r}$。此外，既然这是一个好的单纯形划分，其中任何节点 v 都应满足 $n_v \leq 2n/r$。这两点结合起来说明：在如此选取了 r 之后，上面所给出的 Q(n)递推式的解就是 $O(n^{1/2+\varepsilon})$。❑

对于以上查询时间，你多少会有些失望——毕竟，到目前为止我们所看到的大多数几何数据结构，查询时间要么是 $O(\log n)$，要么是 $\log n$ 的多项式，而不致于像划分树这样高达 $O(\sqrt{n})$左右。很明显，之所以必须付出如此代价，是因为我们想要解决的是（半平面区域计数之类的）真正的二维查询问题。难道，果真就不可能在对数量级的时间内完成这样的查询吗？不是的。本章后面将设计出另一种数据结构，利用它的确可以在对数时间内完成半平面区域查找。不过，该数据结构的查询时间虽然有所改进，但这并不是没有代价的——实际上，它需要占用平方量级的空间。

将这里的方法与第 5 章的区域树以及第 10 章的线段树做一比较，会很有帮助。我们都希望从这些数据结构中，返回关于某组给定几何物体（比如存放在区域树和划分树中的一组点，或者存放在线段树中的一组区间）中某一子集的信息，或者把某一子集完全报告出来。如果通过预处理，能够将在查询中可能出现的所有子集所对应的查找信息在事先就计算出来，那么无论是什么样的查询，都可以很快就找出答案。然而，可能出现的答案往往有很多，从而导致上述构想无法兑现。我们只好采用一种变通的方法——确定一组所谓的正则子集，并在事先计算出这些子集所对应的查询信息。

此后，对于任何一次查询，只需将其对应的答案表示为若干正则子集的无交并集，即可完成查询。我们需要用多少正则子集才能组成所有可能的查询子集，查询时间就大致线性正比于这个数目。区域树和划分树所占用的存储量，将正比于事先计算出来的正则子集总数；线段树所占用的存储量，将正比于事先计算出来的正则子集的规模之和。在查询时间和存储量之间，需要做一个折衷。为保证每一可能的查询都可以表示为尽可能少的正则子集的并集，需要在事先计算出大量的正则子集，这样就需要占用大量的空间。为了降低存储量，需要减少事先计算出来的正则子集数目——然而如此一来，就需要很多个正则子集才能表示一次查询，于是查询时间将会增加。

从二维区域查找问题可以清楚地看出如下现象：按本节算法所构造出来的划分树，只需保存 $O(n)$ 个正则子集，故只占用线性的存储空间；但为了表示落在某一半平面内的所有点，一般都需要使用 $O(\sqrt{n})$个正则子集。实际上，只要保存大约平方量级个正则子集，即可使查询时间降至对数量级。

现在回到待解决的问题——三角形区域查找问题。当待查询区域是三角形而不是半平面时，如果还想使用划分树，需要做哪些调整呢？答案很简单：什么都不用改动。我们仍然可以沿用上面所介绍的数据结构以及相应的查询算法，只要将待查询三角形换成待查询半平面即可。实际上，这种方法对任意（形状）的待查询区域都行之有效。唯一需要讨论的问题是：查询时间将会有何变化？

当查询算法访问到某节点时，其孩子 v 有三种类型：t(v)完全落在待查询区域之内；t(v)完全落在待查询区域之外；以及 t(v)部分落在待查询区域之内。只有对最后一类孩子，才有必要进行递归访问。因此，查询时间就取决于单纯形划分中与待查询区域γ的边界相交的三角形数目。也就是说，必须确定，γ相对于某个单纯形划分而言的穿越数究竟等于多少。对于三角形查找区域而言，这个数很容易确定——在单纯形划分中，一个三角形若与γ的边界相交，则必然与γ的边所在的三条直线（至少）之一相交。既然每条直线最多只能与 $c\sqrt{r}$个三角形相交，故γ的穿越数最多不会超过 $3c\sqrt{r}$。

这样，查询时间的递推式与前面的几乎一样，只需将常数 c 替换为 3c。相应地，只要选取一个更大的 r，就可以使查询时间保持在相同的渐进量级。由此可以得出如下定理：

〖定理 16.4〗

设 S 为平面上任意 n 个点。对任何$\varepsilon > 0$，都存在一个用 $O(n)$空间表示 S 的数据结构（“划分树”），使得对于任一待查询三角形，都可在 $O(n^{1/2+\varepsilon})$时间内统计出 S 中落在其中的点数。另加 $O(k)$时间，还可逐一报告出这些点，其中 k 为报告出来的点数。只要花费 $O(n^{1+\varepsilon})$时间即可构造出该数据结构。

〖证明〗

实际上，我们尚未讨论到的问题只有两个：数据结构的构造时间以及报告输出的时间。

划分树的构造很容易：根据它的递归定义，马上就可以导出一个递归式构造算法。我们将

该算法为n个点构造一棵划分树时所需的时间记作T(n)。给定ε > 0。根据〖定理 16.1〗，对于任何ε' >0，我们都可以在$O(n^{1+\varepsilon'})$时间内构造出S的一个好的单纯形划分，其规模为r，穿越数为$O(\sqrt{r})$。取ε' = ε/2。于是，T(n)将满足如下递推式：

$$T(n) = \begin{cases} O(1) & 若 n = 1 \\ O(n^{1+\varepsilon/2}) + \sum_v T(n_v) & 若 n > 1 \end{cases}$$

其中的和式，取遍该树根节点的所有孩子。既然单纯形划分中的不同类之间不会有公共点，故有$\sum_v n_v = n$，因此该递推式的解为 $T(n) = O(n^{1+\varepsilon})$。

以下将说明：若有 k 个点落在待查询三角形中，则只需另加$O(k)$时间，即可将它们报告出来。这些点都存放在拣出节点之下的各匹叶子中。因此，只要分别遍历（traverse）以每个拣出节点为根节点的子树，即可将它们报告出来。既然树中每一内部节点的度数都不小于 2，内部节点的数目就必与叶子的数目成正比，于是所需的时间必线性正比于被报告出来的点数。❑

16.2 多层划分树

划分树是一种功能强大的数据结构。其长处在于，只需少数的几个分组（即由查询算法拣出的各节点所对应的正则子集），即可确定落在待查询半平面中的所有点。以上所举例子采用划分树结构解决了半平面区域计数问题。就这一问题而言，需要从拣出的正则子集中获得的信息只不过是它们各自的基数。然而在其它的查询问题中，还需要获得正则子集的其它信息，为此必须预先计算出这些数据并将其存储起来。这些需要存放的有关正则子集的信息，并不见得仅是（基数等）一个数字。也可将各正则子集中的元素存储为一个列表、一棵树或你所希望的任一数据结构。这也就是所谓的多层次数据结构（multi-level data structure），它并不是什么新的概念——早在第 5 章，这类结构就已被用于解决矩形区域查找问题；第 10 章中，也曾利用这类结构解决过截窗查询问题。

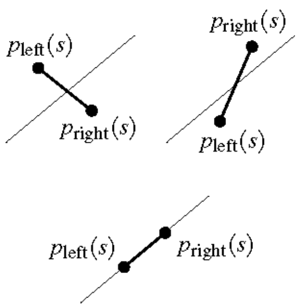


图16-7 线段s与直线l相交的不同情况

以下将介绍一个基于划分树的多层次数据结构。设S为平面上的一组共n条线段，我们想要统计出其中与某一待查询直线（query line）l相交的线段数目。线段s的左、右端点分别记作$p_{left}(s)$和$p_{right}(s)$。如 图 16-7 所示，直线l与s相交，当且仅当s的两个端点分别落在l的两侧（或者s的某个端点正好落在

l上）。下面就来说明，对于那些$p_{right}(s)$落在l上方、$p_{left}(s)$落在l下方的线段$s \in S$，应该如何统计出它们的数目。至于有一个端点正好落在l上的线段，以及$p_{right}(s)$落在l下方、$p_{left}(s)$落在l上方的线段，都可以借助类似的数据结构进行统计。对于直线l垂直的情况，我们约定其左侧为下方，右侧为上方。

算法的构思简明。首先从 S 中找出 $p_{right}(s)$落在 l 上方的所有线段 s。前一节已经介绍了如何借助一棵划分树，从若干个正则子集中拣出这些线段。对这样的每一个正则子集，我们只对其中满足$p_{left}(s)$落在 l 下方的那些线段 s 感兴趣。这个过程也就是一次半平面区域计数查询。只要将每个正则子集都组织为一棵划分树，每次这类查询都可以很快回答。我们再来更加详细地介绍这一方法。这里的数据结构定义如下。对于任一线段集 S'，令 $P_{right}(S') := \{p_{right}(s) \mid s \in S'\}$，即 S'中所有线段的右端点；令 $P_{left}(S') := \{p_{left}(s) \mid s \in S'\}$，即 S'中所有线段的左端点。

- 集合 $P_{right}(S)$被组织为一棵划分树 $\mathcal{T}$。$\mathcal{T}$ 中每个节点 v 所对应的正则子集，记作 $P_{right}(v)$。由$P_{right}(v)$中各右端点分别所属线段所组成的集合，记作 S(v)——即 $S(v) = \{s \mid p_{right}(s) \in P_{right}(v)\}$。（在上下文清楚的情况下，我们有时也直接将 S(v)称为“v 所对应的正则子集”。）
- 在树 $\mathcal{T}$ 中每个节点 v，都相应地存放了集合 $P_{left}(S(v))$，每个这样的集合又被组织为一棵二级划分树 $\mathcal{T}_v^{assoc}$，以支持半平面区域计数查询。这样的一棵划分树，就是节点 v 的联合结构。

借助上述数据结构，即可根据若干个正则子集，从 S 中找出满足“$p_{right}(s)$和 $p_{left}(s)$分别落在 l 上方和下方”的所有线段 s。与之对应的查询算法如下所述。只要将所有被拣出正则子集的基数累加起来，就得到了这类线段的总数。$\mathcal{T}$ 中以 v 为根节点的子树，记作 $\mathcal{T}_v$。

> 算法 SelectIntSegments(l, $\mathcal{T}$)
>
> 输入：待查询直线 l，以及它对应的一棵划分树或子树
>
> 输出：一组正则节点，它们对应于树中与 l 相交的所有线段
>
> 1. $\Upsilon \leftarrow \emptyset$
> 2. **if** ($\mathcal{T}$ 仅由一匹叶子μ组成)
> 3.   **then if** (存放于μ处的那条线段与 l 相交) **then** $\Upsilon \leftarrow \{\mu\}$
> 4.   **else for** ($\mathcal{T}$ 的根节点的每一个孩子)
> 5.   **do if** ($t(v) \subset l^+$)
> 6.   **then** $\Upsilon \leftarrow \Upsilon \cup$ SelectInHalfplane(l^-, $\mathcal{T}_v^{assoc}$)
> 7.   **else if** ($t(v) \cap l \neq \emptyset$)
> 8.   **then** $\Upsilon \leftarrow \Upsilon \cup$ SelectIntSegments(l, $\mathcal{T}_v$)
> 9. **return** Υ

利用上面的这个查询算法，可以找出左、右端点分别落在某一待查询直线下、上方的所有线段。有趣的是，利用这同一棵划分树，也可以找出左、右端点分别落在某一待查询直线上、下方的所有

线段。为此，只需对查询算法稍做调整——交换其中的“l^+”和“l^-”。

如上所述的多层划分树（multi-level partition tree）可以支持拣出相交线段查询（segment intersection selection query），下面就来对该结构做一分析。首先是其占用的存储量。

〖引理 16.5〗

设 S 为平面上的一组共 n 条线段。若借助一棵双层划分树来支持对 S 的拣出相交线段查询，则该结构最多占用 $O(n\log n)$空间。

〖证明〗

第一层划分树中每一正则子集 S(v)的基数，记为 n_v。节点 v 之所以要占用一定的存储空间，是为了存放一棵对应于 S_v 的划分树。由前一节的分析，这只需要线性规模的空间。因此，若用 M(n)来表示对应于 n 条线段的双层划分树需要占用的最大存储空间，则 M(n)满足如下递推式：

$$M(n) = \begin{cases} O(1) & 若 n = 1 \\ \sum_v [O(n_v) + M(n_v)] & 若 n > 1 \end{cases}$$

其中的和式，取遍该树根节点的所有孩子。我们已经知道 $\sum_v n_v = n$，$n_v \le 2n/r$。由于 $r > 2$ 时一个常数，上述关于 M(n)的递推式的解就应该是 $M(n) = O(n\log n)$。 ❑

由上可见，因为在划分树中引入了新的一层，使得占用的存储量增长了一个对数因子。查询时间又会有何变化呢？令人惊异的是，从渐进的角度来看查询时间居然没有任何变化。

〖引理 16.6〗

设 S 为平面上的一组共 n 条线段。对于任何$\varepsilon > 0$，都存在 S 的一棵双层划分树，使得对于任一待查询直线 l，我们都可以从这棵树中拣出 $O(n^{1/2+\varepsilon})$个节点，这些拣出节点所对应的正则子集的无交并集，给出了 S 中与 l 相交的所有线段。为了拣出这些节点，需要花费 $O(n^{1/2+\varepsilon})$时间。这样，我们就可以在 $O(n^{1/2+\varepsilon})$时间内确定相交线段的总数。

〖证明〗

我们再次利用递推式来对查询时间进行分析。任意给定$\varepsilon > 0$。将正则子集S(v)的基数记作n_v。由〖引理 16.3〗可知，可以为每个节点v构造出一个联合结构$\mathcal{T}_v^{assoc}$，使得对$\mathcal{T}_v^{assoc}$的每次查询只需要$O(n_v^{1/2+\varepsilon})$ 的时间。现在考虑S的双层划分树$\mathcal{T}$。在得出这样一棵树时，我们依据的是一个规模为r、穿越数不超过$c\sqrt{r}$的好的单纯形划分，其中取$r := \lceil 2(c\sqrt{2})^{1/\varepsilon} \rceil$。根据〖定理 16.1〗，

这样的一个单纯形划分必然存在。若用Q(n)来表示对应于n条线段的双层划分树的（最大）查询时间，则Q(n)满足如下递推式：

$$Q(n) = \begin{cases} O(1) & 若n = 1 \\ O(rn^{1/2+\varepsilon}) + \sum_{i=1}^{c\sqrt{r}} Q(2n/r) & 若n > 1 \end{cases}$$

按照上面所取的 r 值，解出的 Q(n)为 $O(n^{1/2+\varepsilon})$。由查询时间的这一上界，可以马上得出拣出正则子集数目的上界。 ❑

16.3 切分树

前两节借助划分树解决了平面区域查找问题。就其占用的存储量而言，划分树的确不错——它们只需要大致线性量级的空间。然而，其查询时间却需要 $O(n^{1/2+\varepsilon})$，这已经相当高了。如果使用超过线性规模的存储空间，是否可以使查询时间缩短（比如，到 $O(\log n)$）呢？只要我们还是沿用单纯形划分的方法，就不可能做到这一点——实际上，根本不可能构造出穿越数小于 $O(\sqrt{r})$的单纯形划分；而只要穿越数还是这样大，查询时间就不可能少于 $O(\sqrt{n})$。

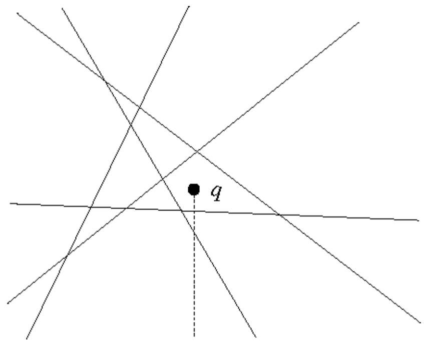


图16-8 半平面区域计数问题在对偶平面中的对应问题：给定一个待查询点，有多少直线位于它的下方？

为找出新的方法，需要从另一角度重新审视这一问题。可采用第 8 章所介绍的对偶变换。第 16.1 节首先讨论的是半平面区域计数问题：给定一组点，统计其中落在某一待查询半平面内的点数。若将这一问题转换到对偶平面，将会得到什么结果？这里假定待查询半平面是正的——亦即，半平面位于其边界直线的上方。在对偶平面中，可做如下设定：给定平面上的一组共n条直线L，对任一待查询点（query point）q，统计出L中位于q下方的直线条数。借助此前各章所给出的工具，不难设计出某种数据结构，使得该问题只需对数量级的查询时间——为此，需要通过观察得出的一条重要结论是：位于待查询点q下方的直线条数，将由q在$A(L)$中所属的那张面唯一确定。因此，可以参照第 6 章所介绍的方法构造出$A(L)$，并对其做预处理，以支持点定位查询；然后，在每张面中记录位于其下方的直线条数。这样，“统计位于任一待查询点下方的直线条数”这一问题，就可以归结为点定

位问题。按照这一方法，需要使用$O(n^2)$的存储空间，而对应的查询时间为$O(\log n)$。

请注意，按此方式，需在事先就将所有可能的查询结果都计算出来——亦即，这里的正则子集已经囊括了所有可能出现的子集。然而，若换成三角形区域计数（triangular range counting）问题，这一方法将不再适用——此时可能出现的三角形太多了，为在事先计算出所有答案，代价太大。因此，可转而尝试递归的方式，通过少量的正则子集之并，来表示位于某一待查询点下方的所有直线。这样，就可以利用前一节所介绍的多层次数据结构，来解决三角形区域查找问题。

这里将用一种名为切分树（cutting tree）的数据结构，构造出完整的一组正则子集。其构思与划分树完全一样：将整个平面划分为多个三角形区域（图 16-9）。不过，这回要求各三角形互不重叠。这一划分，对我们统计出位于某一待查询点下方所有直线有何帮助呢？对任意一组点，为支持三角形区域查找，可通过对偶变换，将其转换为一组直线$L := \{l_1, \dots, l_n\}$。任取划分生成的三角形t，以及不与t相交的直线l_i。若l_i位于t下方（上方），则l_i也必位于t内部任一待查询点的下方（上方）。这就意味着，若待查询点q来自t中，则除了与t相交的那些直线外，其它直线相对q的上下位置关系都是已知的。我们的数据结构将保存划分所生成的全部三角形，并为每个三角形设置一个计数器，指示有多少条直线位于该三角形下方。此外，对每个三角形，还要在事先找出与之相交的所有直线，并递归地将它们组织成同样的结构。在查询该数据结构时，首先找到待查询点q所属的三角形。然后通过递归访问t所对应的子树，统计出在与t相交的直线中有多少条位于q下方。最后，把上述递归调用所统计出来的数目，与位于t下方的直线条数相加。此方法的效率取决于与每个三角形相交的直线条数——此数越小，就只需对越少的直线做递归调用。以下就对这种划分做一形式化描述。

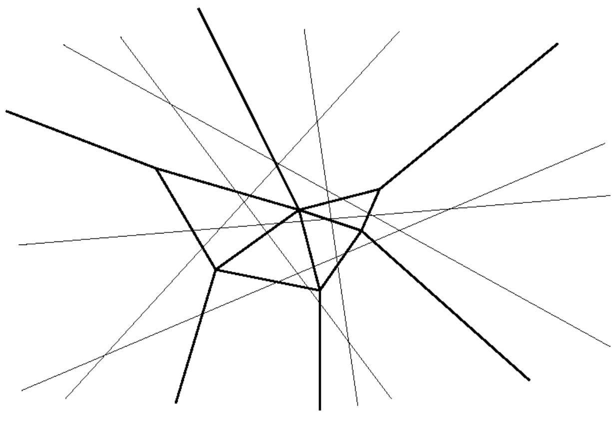


图16-9 六条直线的一个规模为10的(1/2)-切分

设L为平面上n条直线，参数r满足 $1 \le r \le n$。若一条直线与某三角形的内部相交，就称该直线穿越（cross）了此三角形。所谓L的一个$(1/r)$-切分，是由m个互不相交的（可能无界的）三角形组成一个集合$\Xi(L) := \{t_1, \dots, t_m\}$，所有三角形的并集覆盖了整个平面，且其中每个三角形都至多被L中的n/r条直线穿越。所谓切分$\Xi(L)$的规模，即三角形的总数m。图 16-9 就是这种切分的一个例子。

〖定理 16.7〗

对于平面上任意一组共 n 条直线 L，以及满足 $1 \leq r \leq n$ 的任一参数 r，必然存在一个规模为 $O(r^2)$的 (1/r)-切分。而且，可以在 $O(nr)$时间内，构造出这样一个切分（L 中与该切分中各三角形相交的直线，也都被确定而且被存放在对应的三角形中）。

该定理的证明见第 16.4 节提供的参考文献。这里只是关心如何利用这种切分来设计基于这种切分的数据结构——切分树（cutting tree）。对于任何一组共n条直线L，切分树的基本结构如下：

- 若 L 的基数为 1，则对应的切分树本身只是一匹叶子，L 被显式地存放于其中。集合 L 也就是这匹叶子对应的正则子集。
- 否则，该结构就是一棵树 $\mathcal{T}$。在该树根节点的各个孩子，与集合 L 的某一(1/r)-切分中的三角形一一对应，其中 r 为足够大的常数（r 的取法稍后介绍）。L 中位于 t(v)下方、上方的直线所构成的子集，分别称作 v 的下正则子集（lower canonical subset）、上正则子集（upper canonical subset），记作 $L^-(v)$、$L^+(v)$。L 中穿越 t(v)的那些直线所构成的子集，称作 t(v)的穿越子集（crossing subset）。对于任一孩子节点，都要以 v 为根节点，对它的穿越子集递归地定义出一棵划分树，记作 $\mathcal{T}_v$。
- 在每个孩子节点 v 处，都存有与之对应的三角形 t(v)。此外，还要记录一些有关上、下正则子集 $L^+(v)$和 $L^-(v)$的信息——比如，若只要求统计出位于待查询点下方的直线条数，则只需要记录下集合 $L^-(v)$的基数；若是其它的应用，则可能还需要记录下其它的信息。

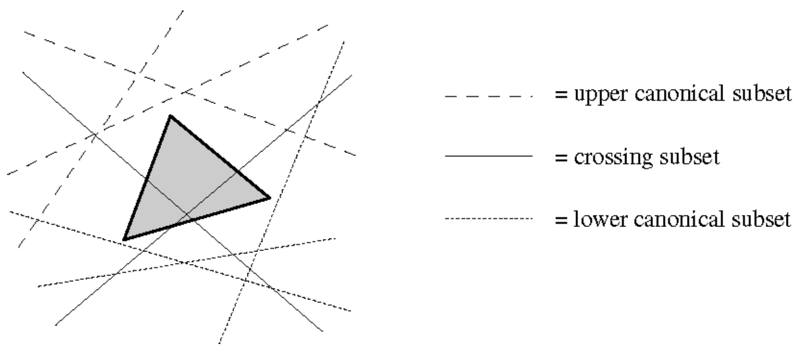


图16-10　下正则子集（短虚线）、上正则子集（长虚线）及穿越子集（实线）[①]

下正则子集、上正则子集及穿越子集的概念如 图 16-10 所示。以下将介绍一个算法，从若干个正则子集中拣出来自于L、位于任一待查询点下方的所有直线。若只要求统计出这些直线的条数，则只需将这些正则子集的基数相加即可。任取一个待查询点q。由所有拣出节点组成的集合，记作ϒ。

① 按照此处定义，上、下正则子集是相对某节点 v 而言的，穿越子集才是相对于三角形 t(v)而言的。——译者

算法 SELECTBELOWPOINT(q, $\mathcal{T}$)
输入：待查询点 q，及其对应的一棵切分树（或子树）
输出：一组正则子集，它们的并集给出了树中位于 q 下方的所有直线
1. $\Upsilon \leftarrow \varnothing$
2. **if** ($\mathcal{T}$ 仅由一匹叶子μ组成)
3. **then if** (存放于μ处的直线位于 q 的下方) **then** $\Upsilon \leftarrow \{\mu\}$
4. **else** **for** ($\mathcal{T}$ 根节点的每个孩子 v)
5. **do** 检查 q 是否落在 t(v)内
6. 令 v_q 为其中满足 $q \in t(v_q)$的那个孩子
7. $\Upsilon \leftarrow \{v_q\} \cup$ SELECTBELOWPOINT(q, $\mathcal{T}_{v_q}$)
8. **return** Υ

〖引理 16.8〗

设 L 为平面上任意一组共 n 条直线。借助切分树结构，对于任一待查询点 q，都可以在 $O(\log n)$时间内拣出 $O(\log n)$个正则子集，而这些正则子集则给出了来自于 L、位于 q 下方的所有直线。故此，可以在 $O(\log n)$时间内统计出这类直线的总数。对于任意的ε > 0，都可以为 L 构造出占用空间不超过 $O(n^{2+\varepsilon})$的一棵切分树。

〖证明〗

将存储 n 条直线的切分树所对应的（最长）查询时间记作 Q(n)。于是 Q(n)必满足如下递推式：

$$Q(n) = \begin{cases} O(1) & 若 n = 1 \\ O(r^2) + Q(n/r) & 若 n > 1 \end{cases}$$

对于任何 r > 1，这个递推式的解都是 $Q(n) = O(\log n)$。

任意给定ε > 0。根据〖定理 16.7〗，我们总是能够为L构造出规模为cr^2的一个(1/r)-切分，其中c是一个常数。我们取$r = \lceil (2c)^{1/\varepsilon} \rceil$，并依照一个(1/r)-切分，构造出一棵这样的切分树。若将这棵树占用的存储量记作M(n)，则M(n)满足如下递推式：

$$M(n) = \begin{cases} O(1) & 若 n = 1 \\ O(r^2) + \sum_v M(n_v) & 若 n > 1 \end{cases}$$

其中的和式，取遍该树根节点的所有孩子。该根节点的孩子总数为 cr^2，而且对于每个孩子 v，都有 $n_v \le n/r$。因此，只要如此取定 r，上述递推式的解必是 $M(n) = O(n^{2+\varepsilon})$。 ❑

由此可以得出结论：借助于一个占用$O(n^{2+\varepsilon})$存储空间的数据结构，我们可以在$O(\log n)$时间内统计出位于任一指定待查询点下方的直线条数。由对偶性，这同时也意味着：我们可以在同样的复杂度内解决半平面区域计数问题。现在，让我们再次考察三角形区域计数问题——给定平面上的一个点集S，统计出其中落在某一待查询三角形内的点数。仿照半平面区域查找的方法，我们也将问题转换到对偶平面中。这样一来，在对偶平面中得到的将是一个什么问题呢？当然，原先的点集肯定会变换为一组直线；然而待查询三角形的对偶呢？这并不是那么一目了然。每个三角形都可以看作是三张半平面的公共交集——也就是说，点p落在该三角形之内，当且仅当p同时落在这三张半平面中。比如在 图 16-11 中，p之所以落在三角形内，是因为$p \in l_1^+$、$p \in l_2^-$和$p \in l_3^-$同时成立。因此，p的对偶直线必然位于l_1^*的下方、l_2^*的上方和l_3^*的上方。

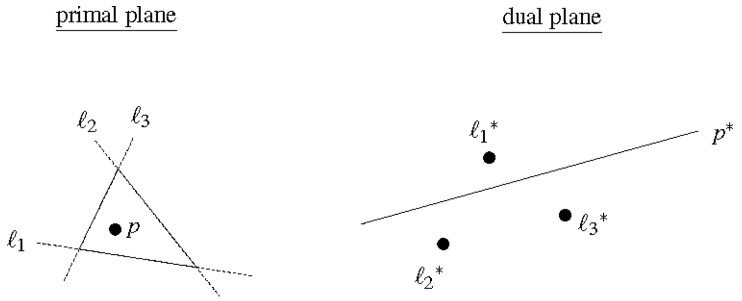


图16-11 三角形区域查找问题：原平面（左），对偶平面（右）

一般而言，三角形区域查找问题可以对偶地表述为：给定平面上的一个直线集 L，以及分别标有“上方”或“下方”记号的三个待查询点 q_1、q_2 和 q_3，统计出 L 中与这三个待查询点的相对位置与其记号完全一致的直线数目。实际上，借助三层切分树结构就可以解决这一问题。不过在这里，我们只针对一个稍微简单一点的问题，来介绍相应的数据结构。这个问题是：给定一个直线集 L 以及一对待查询点 q_1 和 q_2，从 L 中找出同时位于这两个待查询点下方的所有直线。我们将证明，的确可以借助一棵双层切分树来解决这一问题。只要掌握了这种方法，就不难自行设计出一种三层切分树结构，来解决三角形区域查找问题的对偶问题。

给定由平面上 n 条直线组成的一个集合 L，为了能够从中找出同时位于一对待查询点 q_1 和 q_2 下方的所有直线，需要借助一棵定义如下的双层切分树：

- 集合 L 被组织为一个切分树 T。
- 在树 T 第一层的每个节点 v 处，分别将 v 的下正则子集组织成一棵第二层树 T_v^{assoc}。

这里的构思是：借助树的第一层拣出若干个正则子集，这些子集（的并集）给出了位于 q_1 下方的所有里直线。接下来，再借助分别存储了这些被拣出正则子集的各联合结构（即第二层的树），从中拣出位于 q_2 下方的所有直线。既然这些联合结构都是单层切分树，故可以应用 SELECTBELOWPOINT 算法对其进行查询。完整的查询算法可以描述如下：

算法 SELECTBELOWPAIR(q_1, q_2, $\mathcal{T}$)

输入：一对待查询点 q_1 和 q_2，以及一棵切分树（或它的子树）

输出：一组正则子集，它们给出了同时位于 q_1 和 q_2 下方的所有直线

1. $\Upsilon \leftarrow \varnothing$
2. **if** ($\mathcal{T}$ 仅由一匹叶子μ组成)
3. **then if** (存放在μ处的直线同时位于 q_1 和 q_2 的下方) **then** $\Upsilon \leftarrow \{\mu\}$
4. **else for** (树 $\mathcal{T}$ 根节点的每一个孩子 v)
5. **do** 检查 q_1 是否落在 t(v)内
6. 令 v_{q_1} 为其中满足 $q_1 \in t(v_{q_1})$ 的那个孩子
7. $\Upsilon_1 \leftarrow$ SELECTBELOWPOINT(q_2, $\mathcal{T}^{assoc}_{q_1}$)
8. $\Upsilon_2 \leftarrow$ SELECTBELOWPAIR(q_1, q_2, $\mathcal{T}_{v_{q1}}$)
9. $\Upsilon \leftarrow \Upsilon_1 \cup \Upsilon_2$
10. **return** Υ

你应该还记得：在划分树中增加一个层次，并不会增加查询时间，但占用的存储量却会增加一个对数因子。而对切分树来说，情况正好颠倒过来——在层次增加之后，其查询时间将增加一个对数因子，而其占用的存储量却保持不变。这可以归纳为如下引理：

〖引理 16.9〗

设 L 为平面上任意一组共 n 条直线。借助一棵双层切分树，对于任何一对待查询点，我们都可以在 $O(\log^2 n)$时间内拣出 $O(\log^2 n)$个正则子集，这些子集给出了来自于 L、同时位于这两个待查询点下方的所有直线。故此，可以在 $O(\log^2 n)$时间内统计出这些直线的数目。对于任何$\varepsilon > 0$，都可以构造出占用 $O(n^{2+\varepsilon})$存储空间的这样一棵双层切分树。

〖证明〗

将存储n条直线的双层切分树所对应的（最长）查询时间记作Q(n)。既然其中的每个联合结构都是一棵单层切分树，故根据〖引理 16.8〗，对它们的查询时间就应该是$O(\log n)$。于是，Q(n)必满足如下递推式：

$$Q(n) = \begin{cases} O(1) & 若 n = 1 \\ O(r^2) + O(\log n) + Q(n/r) & 若 n > 1 \end{cases}$$

只要取常数 r > 1，这个递推式的解就必然是 $Q(n) = O(\log^2 n)$。

任意取定$\varepsilon > 0$。再次根据〖引理 16.8〗，可以为根节点的每个孩子分别构造出一个联合结构，而且这些联合结构总共只占用$O(n^{2+\varepsilon})$的存储空间。因此，若将切分树所占用的存储量记

作M(n)，则M(n)必满足如下递推式：

$$M(n) = \begin{cases} O(1) & 若n = 1 \\ \sum_v [O(n^{2+\varepsilon}) + M(n_v)] & 若n > 1 \end{cases}$$

其中的和式，取遍该树根节点的所有孩子 v。根节点的孩子总数为 $O(r^2)$，而且每个孩子 v 都满足 $n_v \le n/r$。由此，只要取足够大的常数 r，上述递推式的解必然是 $M(n) = O(n^{2+\varepsilon})$。（至此，你或许多少会感到有些乏味。果真如此，就说明你的感觉很对——实际上，无论是切分树、划分树还是它们的多层次变种，都可以套用这种分析方法。） ❑

至此，我们已经针对“拣出同时位于一对待查询点下方的所有直线（或对它们进行计数）”这一问题，设计了一种双层切分树结构，并对其性能做了分析。如果换成三角形区域查找问题，我们就需要使用一棵三层切分树。这种三层切分树的设计过程及其分析方法，完全可以照搬双层切分树的模式。因此，不用费多大气力，你就应该能证明如下定理：

〖定理 16.10〗

设 S 为平面上任意一组共 n 个点。对于任何$\varepsilon > 0$，都存在 S 的一个数据结构（称作切分树），它只占用 $O(n^{2+\varepsilon})$的存储空间；借助这一结构，对任一待查询三角形，我们都能在 $O(\log^3 n)$时间内统计出来自于 S、落在该三角形内的点数。只要另花费 $O(k)$时间，就可以逐一报告出这些点，其中 k 为实际被报告出来的点数。可以在 $O(n^{2+\varepsilon})$时间内构造出这样一个数据结构。

实际上，还有性能略好于该定理的方法。我们将在第 16.4 节和 习题部分继续讨论这一问题。

16.4 注释及评论

区域查找问题，是计算几何（computational geometry）领域被研究得最为深入的问题之一。我们可以将这一问题划分为正交区域查找（orthogonal range searching）和单纯形区域查找（simplex range searching）两类。正交区域查找是第 5 章讨论的主题。本章讨论了单纯形区域查找在平面上的变种——三角形区域查找问题。下面，将就单纯形区域查找的研究历史做一扼要回顾，并就本章正文所介绍理论在高维空间中的变型做一讨论。

我们首先介绍了只需要大约线性量级存储空间的一种数据结构，借助它可解决平面上的单纯形区域查找问题。该结构由Willard[388]首先提出，其构思与本章的划分树一样——将整个平面划分为若干子区域。不过，他当初所采用的划分的穿越数太大，致使其数据结构的查询时间长达$O(n^{0.774})$。

随着更好的单纯形划分不断被提出来，效率更高的划分树也成为了可能 [111][169][209][394]。采用与划分树略微不同的另一数据结构——穿刺数很小的支撑树 [384][112]——也可以进一步提高性能。截至目前，解决三角形区域查找问题的最好方法是由Matousek[263]提出的。在他的那篇论文中，给出了〖定理 16.1〗的证明。Matousek还提出了一种更为复杂的数据结构，其查询时间为$O(\sqrt{n}\cdot 2^{o(\log^* n)})$。不过，以这种数据结构为基础建立多层次树结构却不太容易。

$\mathbb{R}^d$中的单纯形区域查找问题可以表述为：将$\mathbb{R}^d$中的点集S预处理为某种数据结构，使得S中落在任一待查询单纯形内的点，都可被高效地计数或报告出来。Matousek还证明了一些有关高维单纯形划分的结论。$\mathbb{R}^d$中单纯形划分的定义，与平面情形类似。唯一的区别在于：平面上的三角形换成了d-单纯形，穿越数则是相对于超平面（而不再是直线）定义的。Matousek还证明：对于$\mathbb{R}^d$中的任一点集，都存在一个规模为r、穿越数为$O(r^{1-1/d})$的单纯形划分。基于这样一个单纯形划分，对于任何$\varepsilon > 0$，都可以构造出一棵只占用线性存储空间的划分树；借助这棵树，可以在$O(n^{1-1/d+\varepsilon})$的查询时间内完成单次单纯形区域查找。查询时间还可以进一步改进为$O(n^{1-1/d}\cdot(\log n)^{O(1)})$。Matousek的数据结构所对应的查询时间，已经与Chazelle[89]所证明的下界（lower bound）非常接近。按照Chazelle的结论，支持三角形区域查找问题的任何一种数据结构，若其占用的存储量为$O(m)$，则其对应的查询时间必然不会少于$\Omega(n/(m^{1/d}\cdot\log n))$。这样，对于需要占用线性存储空间的任何数据结构，查询时间都不可能少于$\Omega(n^{1-1/d}/\log n)$。（就平面情况而言，已经证明了一个更紧的下界——$\Omega(n/\sqrt{m})$。）

在对数查询时间内解决单纯形区域查找问题的数据结构，一直被人们广泛关注。Clarkson[131]首先意识到：可以基于切分（cutting）的概念，来设计支持区域查找的数据结构。他通过概率分析的方法证明：$\mathbb{R}^d$中的任一超平面集，都存在规模为$O(r^d)$的一个$O(\log r/r)$-切分。利用这一结论，他还提出了支持半空间区域查询（half-space range query）的一种数据结构。此后，还有一些人对该结果做了改进，并设计出实现切分的有效算法。截至目前，最好的此类算法是由Chazelle[95]提出的。他证明：对于任何参数r，都可能通过一个确定性算法（deterministic algorithm）在$O(nr^{d-1})$时间内构造出规模为$O(r^d)$的一个$(1/r)$-切分。基于这种切分，可设计出一种多层切分树（multi-level cutting tree）结构，以解决单纯形区域查找问题——就像本章对平面情况的处理一样。如此得出的数据结构需要占用$O(n^{d+\varepsilon})$的存储空间，其查询时间为$O(\log^d n)$。这一查询时间可降至$O(\log n)$。得益于Chazelle切分的一个特殊性质，甚至还可将存储量的$O(n^\varepsilon)$因子去除掉 [265]——然而在如此改造之后，该数据结构的查询时间将无法再降低到$O(\log n)$。同样地，这些结果也都与Chazelle证明的下界非常接近。

只要将划分树和切分树适当地结合起来，就可以得到存储量介于划分树和切分树之间的一种数据结构。具体来说，对于任何$n \le m \le n^d$，这种数据结构的规模为$O(m^{1+\varepsilon})$，查询时间为$O(n^{1+\varepsilon}/m^{1/d})$——这与下界已经非常接近（具体方法，请参见 习题 16.16）。

以上重点讨论了单纯形区域查找问题。当然，半空间区域查找是其中一个特例。但研究结果表

明，针对这一特殊问题可得到更好的结果。例如平面上的半平面区域报告（half-plane range reporting，不是计数）问题，存在一个占用存储量为$O(n)$、查询时间为$O(\log n + k)$的数据结构 [107]。这里，k为实际被报告出来的点数。对高维问题，也可同样进行改进——借助一种占用$O(n\log\log n)$存储空间的数据结构，可在$O(n^{1-1/\lfloor d/2\rfloor}\cdot(\log n)^{O(1)} + k)$时间内报告出落在任一待查询半空间内的所有点 [264]。

最后，Agarwal 和 Matousek 还将有关区域查找的结果，推广到半代数集（semi-algebraic set）的待查询区域。

16.5 习题

习题 16.1 设 S 为平面上任意一组共 n 个点。

a. 假定 S 中各点都落在$\sqrt{n}\times\sqrt{n}$的格子上。（为简化起见，假定 n 是一个平方数。）设参数 r 满足 $1 \le r \le n$。试绘出 S 的一个单纯形划分，要求其规模为 r，穿越数不超过 $O(\sqrt{r})$。

b. 假设 S 中所有点都共线。试绘出 S 的另一个规模为 r 的单纯形划分。该单纯形划分的穿越数为多少？

习题 16.2 试证明：从划分树中被拣出的节点，正好就是具有如下性质的那些节点 v：$t(v) \subset h$（或者，当 v 为一匹叶子时，存放 v 处的那个节点 v 落在 h 内），而且 v 的任何祖先μ都不会满足 $t(\mu) \subset h$。试利用这一性质证明：这些拣出节点所对应正则子集的无交并集，就是 $S \cap h$。

习题 16.3 试证明：〖引理 16.2〗的证明中关于$M(n)$的递推式，解为$M(n) = O(n)$。

习题 16.4 试证明：〖引理 16.3〗的证明中关于$Q(n)$的递推式，解为$Q(n) = O(n^{1/2+\varepsilon})$。

习题 16.5 假设我们已经按照第 425 页的定义，构造出了一棵划分树。不过，该划分树并不见得是基于某个好的单纯形划分构造出来的。如果是这样，划分树所占用的存储量有何变化？查询时间呢？

习题 16.6 〖引理 16.3〗指出：对任何$\varepsilon > 0$，只要将确定该树分支度的参数r取为一个足够大的常数，就总是可以构造出查询时间为$O(n^{1/2+\varepsilon})$的一棵划分树。如果根据n来选取r，甚至还能做到更好。试说明：只要取$r = \sqrt{n}$，即可使查询时间降至$O(\sqrt{n}\cdot\log n)$。（请注意：按此方法，即使是在同一棵树中，不同节点所对应的r值也将会不同。不过，这算不上是个问题。）①

习题 16.7 试证明：〖引理 16.5〗的证明中关于$M(n)$的递推式，解为$M(n) = O(n\log n)$。

① 本书第二版的勘误指出，正确结论的应该是：可以将查询时间降为 $O(\sqrt{n}\cdot\log^c n)$（而不是 $O(\sqrt{n}\cdot\log n)$），其中 c 为一个适当的常数。可惜第三版中依然未予更正。——译者

习题 16.8 试证明：〖引理 16.6〗的证明中关于Q(n)的递推式，解为$Q(n) = O(n^{1/2+\varepsilon})$。

习题 16.9 设 T 为平面上任意一组共 n 个三角形。所谓的逆向区域计数查询（inverse range counting query），就是从T中找出包含某一待查询点的所有三角形。

a. 试给出支持逆向区域计数查询的一种数据结构,要求它占用的存储量应大致为线性量级（比如，对某个常数 c，为 $O(n\log^c n)$）。并对该数据结构的存储空间及查询时间做一分析。

b. 如果在事先已经确定所有三角形互不相交，是否能做到更好?

习题 16.10 设 L 为平面上任意一组共 n 条直线。

a. 假定 L 中含有$\lfloor n/2\rfloor$条垂线、$\lceil n/2\rceil$条水平线。设参数 r 满足 $1 \le r \le n$。试绘出 L 的一个规模为 $O(r^2)$的(1/r)-切分。

b. 再假定 L 中的所有直线都是垂直的。试绘出 L 的另一个(1/r)-切分。这个切分的规模有多大?

习题 16.11 试证明:〖引理 16.8〗的证明中关于Q(n)和M(n)的两个递推式,解分别为$Q(n) = O(\log n)$和$M(n) = O(n^{2+\varepsilon})$。

习题 16.12 试证明：〖引理 16.9〗的证明中关于Q(n)和M(n)的两个递推式，解分别为$Q(n) = O(\log^2 n)$和$M(n) = O(n^{2+\varepsilon})$。

习题 16.13 对双层切分树进行查询时，需沿着树中的一条路径，逐一访问各节点所对应的联合结构。根据〖引理 16.8〗，对于存有m条直线的联合结构，查询时间为$O(\log m)$。由于主树深度为$O(\log n)$，故总的查询时间为$O(\log^2 n)$。若能够将主切分树的参数r选为大于常数（比如n^δ，其中$\delta > 0$），则可降低主树深度，从而缩短查询时间。然而遗憾的是，〖引理 16.9〗的证明所给出的关于Q(n)的递推式中，含有一个$O(r^2)$项。

a. 试说明如何才能避开这一问题，使得我们可以选取$r := n^\delta$。

b. 试证明：该双层数据结构的查询时间为$O(\log n)$。

c. 试证明：该数据结构占用的存储量为$O(n^{2+\varepsilon})$。

习题 16.14 试设计一种数据结构，在$O(\log^3 n)$时间内解决三角形区域查找问题。请对该数据结构及其对应的查询算法做详细的描述，并对其存储空间及查询时间性能做一分析。

习题 16.15 设 S 为平面上任意 n 个点，其中每个点都被赋予一个正的实数权值。试给出两种数据结构，分别解决如下查询问题：对于任一指定的待查询半平面，找出落在其中的权值最大点。要求前一数据结构只能占用线性的存储空间，后一数据结构的查询时间不得超过对数量级。针对这两种数据结构的存储空间及查询时间性能，试分别做一分析。

习题 16.16 本章为了解决半平面区域查找问题，采用了一种只占用线性存储空间的数据结构（即划分树），不过它的查询时间很长；我们还采用了另一种数据结构（即切分树），它的查询时间不超过对数量级，但却需要占用更多的空间。有时，我们可能会希望找出一种介于二者之间的数据结构——其查询时间比划分树短，而占用的存储空间却少于

切分树。本题就是要看看应该如何设计出这样一种数据结构。

假设可用空间量只有 $O(m^{1+\varepsilon})$，其中 m 介于 n 和 n^2 之间。于是，我们的目标就可以表述为：构造一种数据结构，以支持从半平面中挑选点的查询，该结构占用的空间为 $O(m^{1+\varepsilon})$，而查询时间要尽可能地短。我们的构思是，从已知速度最快的一种数据结构（切分树）开始，若存储空间消耗殆尽，则转向速度慢些的数据结构（划分树）。亦即，不断递归地构造一棵切分树，直到其所需存储的直线数目少于某一阈值 $\hat{n}$。

a. 试就该数据结构及其查询算法做一详细描述。

b. 试计算出一个适当的阈值 $\hat{n}$，使得总的存储量为 $O(m^{1+\varepsilon})$。

c. 试对该数据结构的查询时间做一分析。

参考文献

[1] P. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 36. Chapman & Hall/CRC, 2004.

[2] P. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. Goodman, and R. Pollack, editors, Advances in Discrete and Computational Geometry, pages 1-56. American Mathematical Society, 1998.

[3] P. Agarwal, J. Pach, and M. Sharir. State of the union (of geometric objects): A review. In J. Goodman, J. Pach, and R. Pollack, editors, Computational Geometry: Twenty Years Later. American Mathematical Society, 2007.

[4] P K. Agarwal. Partitioning arrangements of lines II: Applications. Discrete Comput. Geom., 5:533-573, 1990.

[5] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. Discrete Comput. Geom., 28:291-312,2002.

[6] P. K. Agarwal, M. de Berg, J. Matousek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. SIAM J. Comput., 27:654-667, 1998.

[7] P. K. Agarwal and M. van Kreveld. Implicit point location in arrangements of line segments, with an application to motion planning. Internat. J. Comput. Geom. Appl., 4:369-383, 1994.

[8] P. K. Agarwal and J. Matousek. On range searching with semialgebraic sets. Discrete Comput. Geom., 11:393-418, 1994.

[9] P K. Agarwal and M. Sharir. Efficient randomized algorithms for some geometric optimization problems. Discrete Comput. Geom., 16:317-337, 1996.

[10] A. Aggarwal. The art gallery problem: Its variations, applications, and algorithmic aspects. Ph.D. thesis, Johns Hopkins Univ., Baltimore, MD, 1984.

[11] A. Aggarwal, L. J. Guibas, J. B. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. Discrete Comput. Geom., 4:591-604,1989.

[12] O. AichhoIzer, E Aurenhammer, S.-W Chen, N. Katoh, M. Taschwer, G. Rote, and Y-E Xu. Triangulations intersect nicely. Discrete Comput. Geom., 16:339-359, 1996.

[13] V. Akman. Unobstructed Shortest Paths in Polyhedral Environments. Lecture Notes in Computer Science, vol. 251. Springer-Verlag, 1987.

[14] S. Aluru. Quadtrees and octrees. In D. Metha and S. Sahni, editors, Handbook of Data Structures and Applications, chapter 19. Chapman & Hall/CRC, 2005.

[15] N. M. Amato, M. T. Goodrich, and E. A. Ramos. A randomized algorithm for triangulating a simple polygon in linear time. Discrete Comput. Geom., 26:245-265.2001.

[16] N. Amenta. Helly-type theorems and generalized linear programming. Discrete Comput. Geom., 12:241-261, 1994.

[17] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. Inform. Process. Lett., 9:216-219, 1979.

[18] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In SIGMOD Conf., pages 347-358, 2004.

[19] L. Arge, G. Brodal, and L. Georgiadis. Improved dynamic planar point location. In Proc. 47th Annu. IEEE Sympos. Found. Comput. Sci., pages 305-314, 2006.

[20] E. M. Arkin, J. S. B. Mitchell, and S. Suri. Logarithmic-time link path queries in a simple polygon. Internat. J. Comput. Geom. Appl., 5:369-395, 1995.

[21] B. Aronov, M. de Berg, and C. Gray. Ray shooting and intersection searching amidst fat convex polyhedra in 3-space. In Proc. 22nd Annu. ACM Sympos. Comput. Geom., pages 88-94, 2006.

[22] B. Aronov and M. Sharir. On translational motion planning of a convex polyhedron in 3-space. SIAM J. Comput., 26:1785-1803, 1997.

[23] T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. Algorithmica, 1:49-63, 1986.

[24] T. Asano, D. Kirkpatrick, and C. K. Yap. dl -optimal motion for a rod. In Proc. 12th Annu. ACM Sympos. Comput. Geom., pages 252-263, 1996.

[25] E Aurenhammer. A criterion for the affine equality of cell complexes in $\mathbb{R}^d$ and convex polyhedra in $\mathbb{R}^{d+1}$. Discrete Comput. Geom., 2:49-64, 1987.

[26] F. Aurenhammer. Power diagrams: properties, algorithms and applications. SIAM J. Comput., 16:78-96, 1987.

[27] E Aurenhammer. Linear combinations from power domains. Geom. Dedicata, 28:45-52, 1988.

[28] E Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. ACM Comput. Surv., 23:345-405, 1991.

[29] E Aurenhammer and H. Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. Pattern Recogn., 17:251-257, 1984.

[30] F. Aurenhammer, E Hoffmann, and B. Aronov. Minkowski-type theorems and least-squares clustering. Algorithmica, 20:61-76, 1998.

[31] E Aurenhammer and O. Schwarzkopf. A simple on-line randomized incremental algorithm for computing higher order Voronoi diagrams. Internat. J. Comput. Geom. Appl., 2:363-381, 1992.

[32] D. Avis and G. T. Toussaint. An efficient algorithm for decomposing a polygon into star-shaped polygons. Pattern Recogn., 13:395-398, 1981.

[33] F. Avnaim, J.-D. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles. In Proc. 5th IEEE Internat. Conf. Robot. Autom., pages 1656-1661, 1988.

[34] C. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. SIAM J. Comput., 21:339-364, 1992.

[35] I. J. Balaban. An optimal algorithm for finding segment intersections. In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 211-219, 1995.

[36] C. Ballieux. Motion planning using binary space partitions. Technical Report InfIsrc/93-25, Utrecht University, 1993.

[37] B. Barber and M. Hirsch. A robust algorithm for point in polyhedron. In Proc. 5th Canad. Conf. Comput. Geom., pages 479-484, Waterloo, Canada, 1993.

[38] R. E. Barnhill. Representation and approximation of surfaces. In J. R. Rice, editor, Math. Software III, pages 69-120. Academic Press, New York, 1977.

[39] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. Internat. J. Robot. Res., 10:628-649, 1991.

[40] B. G. Baumgart. A polyhedron representation for computer vision. In Proc. AFIPS Natl. Comput. Conf., vol. 44, pages 589-596, 1975.

[41] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. J. Algorithms, 17:342-380, 1994.

[42] P. Belleville, M. Keil, M. McAllister, and J. Snoeyink. On computing edges that are in all minimum-weight triangulations. In Proc. 12th Annu. ACM Sympos. Comput. Geom., pages V7-V8, 1996.

[43] R. V. Benson. Euclidean Geometry and Convexity. McGraw-Hill, New York, 1966.

[44] J. L. Bentley. Multidimensional binary search trees used for associative searching. Commun. ACM, 18:509-517, 1975.

[45] J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.

[46] J. L. Bentley. Decomposable searching problems. Inform. Process. Lett., 8:244-251, 1979.

[47] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput., C-28:643-647, 1979.

[48] J. L. Bentley and J. B. Saxe. Decomposable searching problems 1: Static-to-dynamic transformation. J. Algorithms, 1:301-358, 1980.

[49] M. Berg. Vertical ray shooting for fat objects. In Proc. 21st Annu. ACM Sympos. Comput. Geom., pages 288-295, 2005.

[50] M. de Berg. Computing half-plane and strip discrepancy of planar point sets. Comput. Geom. Theory Appl., 6:69-83, 1996.

[51] M. de Berg. Linear size binary space partitions for uncluttered scenes. Algorithmica, 28:353-366, 2000.

[52] M. de Berg, P. Bose, D. Bremner, S. Ramaswami, and G. T. Wilfong. Computing constrained minimum-width annuli of point sets. Comput.-Aided Design, 30:267-275,1998.

[53] M. de Berg and C. Gray. Vertical ray shooting and computing depth orders for fat objects. In Proc. 17th ACM-SIAM Sympos. Discrete Algorithms, pages 494-503, 2006.

[54] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. Comput. Geom. Theory Appl., 8:317-333, 1997.

[55] M. de Berg, M. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. Algorithmica, 34:81-97, 2002.

[56] M. de Berg, M. van Kreveld, B. J. Nilsson, and M. H. Overmars. Shortest path queries in rectilinear worlds. Internat. J. Comput. Geom. Appl., 2:287-309, 1992.

[57] M. de Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. J. Algorithms, 18:256-277, 1995.

[58] M. de Berg, J. Matousek, and O. Schwarzkopf. Piecewise linear paths among convex obstacles. Discrete Comput. Geom., 14:9-29, 1995.

[59] M. de Berg and O. Schwarzkopf. Cuttings and applications. Internat. J. Comput. Geom. Appl., 5:343-355, 1995.

[60] M. de Berg and M. Streppel. Approximate range searching using binary space partitions. Comput. Geom. Theory Appl., 33:139-151, 2006.

[61] M. de Berg and M. Streppel. Approximate range searching using binary space partitions. Comput. Geom. Theory Appl., 33:139-151, 2006.

[62] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and E K. Hwang, editors, Computing in Euclidean Geometry. Lecture Notes Series on Computing, vol. 1, pages 23-90. World Scientific, Singapore, 1992.

[63] M. Bern and D. Eppstein. Polynomial-size nonobtuse triangulation of polygons. Internat. J. Comput. Geom. Appl., 2:241-255, 449-450, 1992.

[64] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. J. Comput. Syst. Sci., 48:384-409, 1994.

[65] M. Bern and P. Plasman. Mesh generation. In J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry, 2nd edn., chapter 6. Elsevier, 1999.

[66] M. W. Bern, H. Edelsbrunner, D. Eppstein, S. L. Mitchell, and T. S. Tan. Edge insertion for optimal triangulations. Discrete Comput. Geom., 10:47-65, 1993.

[67] M. Bern, S. Mitchell, and J. Ruppert. Linear-size nonobtuse triangulation of polygons. Discrete Comput. Geom., 14:411-428, 1995.

[68] B. K. Bhattacharya and J. Zorbas. Solving the two-dimensional findpath problem using a line-triangle representation of the robot. J. Algorithms, 9:449-469, 1988.

[69] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. Discrete Comput. Geom., 8:51-71, 1992.

[70] J.-D. Boissonnat, O. Devillers, and M. Teillaud. A semidynamic construction of higher-order Voronoi diagrams and its randomized analysis. Algorithmica, 9:329-356, 1993.

[71] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. Theoret. Comput. Sci., 112:339-354, 1993.

[72] P. Bose and G. Toussaint. Geometric and computational aspects of manufacturing processes. Comput. & Graphics, 18:487-497, 1994.

[73] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In Proc. 43rd Annu. IEEE Sympos. Found. Comput. Sci., pages 617-626, 2002.

[74] R. A. Brooks and T. Lozano-P6rez. A subdivision algorithm in configuration space for findpath with rotation. IEEE Trans. Syst. Man Cybern., 15:224-233, 1985.

[75] G. Brown. Point density in stems per acre. New Zealand Forestry Service Research Notes, 38:1-11, 1965.

[76] J. L. Brown. Vertex based data dependent triangulations. Comput. Aided Geom. Design, 8:239-251, 1991.

[77] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". IEEE Trans. Comput., C-30:147-148, 1981.

[78] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In Proc. 5th ACM-SIAM Sympos. Discrete Algorithms, pages 16-23, 1994.

[79] A. Bykat. Convex hull of a finite set of points in two dimensions. Inform. Process. Lett., 7:296-298, 1978.

[80] J. Canny. The Complexity of Robot Motion Planning. MIT Press, Cambridge, MA, 1987.

[81] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In Proc. 8th Annu. ACM Sympos. Comput. Geom., pages 251-260,1992.

[82] T. M. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. Discrete Comput. Geom., 16:369-387, 1996.

[83] T. M. Chand and S. S. Kapur. An algorithm for convex polytopes. J. ACM, 17:78-86, 1970.

[84] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. J. ACM, 17:78-86, 1970.

[85] B. Chazelle. A theorem on polygon cutting with applications. In Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci., pages 339-349, 1982.

[86] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. SIAM J. Comput., 13:488-507, 1984.

[87] B. Chazelle. Filtering search: a new approach to query-answering. SIAM J. Comput., 15:703-724, 1986.

[88] B. Chazelle. Reporting and counting segment intersections. J. Comput. Syst. Sci., 32:156-182, 1986.

[89] B. Chazelle. Lower bounds on the complexity of polytope range searching. J. Amer. Math. Soc., 2:637-666, 1989.

[90] B. Chazelle. Lower bounds for orthogonal range searching, I: the reporting case. J. ACM, 37:200-212,1990.

[91] B. Chazelle. Lower bounds for orthogonal range searching, II: the arithmetic model. J. ACM, 37:439-463, 1990.

[92] B. Chazelle. Triangulating a simple polygon in linear time. In Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci., pages 220-230, 1990.

[93] B. Chazelle. An optimal convex hull algorithm and new results on cuttings. In Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci., pages 29-38, 1991.

[94] B. Chazelle. Triangulating a simple polygon in linear time. Discrete Comput. Geom., 6:485-524, 1991.

[95] B. Chazelle. Cutting hyperplanes for divide-and-conquer. Discrete Comput. Geom., 9:145-158, 1993.

[96] B. Chazelle. Geometric discrepancy revisited. In Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci., pages 392-399, 1993.

[97] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. Discrete Comput. Geom., 10:377-409, 1993.

[98] B. Chazelle and H. Edelsbrunner. An improved algorithm for constructing kth-order Voronoi diagrams. IEEE Trans. Comput., C-36:1349-1354, 1987.

[99] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In Proc. 29th Annu. IEEE Sympos. Found. Comput. Sci., pages 590-600, 1988.

[100] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. J. ACM, 39:1-54, 1992.

[101] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Report UWC1)CS-R-901578, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1989.

[102] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. Theoret. Comput. Sci., 84:77-105, 1991.

[103] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Algorithmica, 11: 116-1321. 1994.

[104] B. Chazelle and J. Friedman. Point location among hyperplanes and unidirectional ray-shooting. Comput. Geom. Theory Appl., 4:53-62, 1994.

[105] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. Algorithmica, 1:133-162, 1986.

[106] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. Algorithmica, 1:163-191, 1986.

[107] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. BIT, 25:76-90, 1985.

[108] B. Chazelle and J. Incerpi. Triangulating a polygon by divide and conquer. In Proc. 21st Allerton Conf. Commun. Control Comput., pages 447-456, 1983.

[109] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. ACM Trans. Graph., 3:135-152, 1984.

[110] B. Chazelle and L. Palios. Triangulating a non-convex polytope. Discrete Comput. Geom., 5:505-526, 1990.

[111] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. Algorithmica, 8:407-429, 1992.

[112] B. Chazelle and E. We1A. Quasi-optimal range searching in spaces of finite VC-dimension. Discrete Comput. Geom., 4:467-489, 1989.

[113] D. Z. Chen, K. S. Klenk, and H.-Y. T. Tu. Shortest path queries among weighted obstacles in the rectilinear plane. In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 370-379, 1995.

[114] Y-13. Chen and D. Ierardi. Time-optimal trajectories of a rod in the plane subject to velocity constraints. Algorithmica, 18:165-197, June 1997.

[115] S. W. Cheng and R. Janardan. New results on dynamic planar point location. SIAM J. Comput., 21:972-999, 1992.

[116] L. P. Chew. Building Voronoi diagrams for convex polytopes in linear expected time. Technical Report PCS-TR90-147, Dept. Math. Comput. Sci., Dartmouth College, Hanover, NH, 1986.

[117] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In Proc. 9th Annu. ACM Sympos. Comput. Geom., pages 274-280, 1993.

[118] L. P. Chew and R. L. Drysdale, III. Voronoi diagrams based on convex distance functions. In Proc. 1st Annu. ACM Sympos. Comput. Geom., pages 235-244, 1985.

[119] L. P. Chew and K. Kedem. A convex polygon among polygonal obstacles: Placement and high-clearance motion. Comput. Geom. Theory Appl., 3:59-89, 1993.

[120] Y.-J. Chiang, F. R Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. SIAM J. Comput., 25:207-233, 1996.

[121] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. Proc. IEEE, 80:1412-1434. September 1992.

[122] Y.-J. Chiang and R. Tamassia. Optimal shortest path and minimum-link path queries between two convex polygons inside a simple polygonal obstacle. Internat. J. Comput. Geom. Appl., 7:85-121, 1997.

[123] E Chin, J. Snoeyink, and C-A. Wang. Finding the medial axis of a simple polygon in linear time. In Proc. 6th Annu. Internat. Sympos. Algorithms Comput. (ISAAC 95). Lecture Notes in Computer Science, vol. 1004, pages 382-391. Springer-Verlag, 1995.

[124] N. Chin and S. Feiner. Near real time shadow generation using bsp trees. In Proc. SIGGRAPH '89, pages 99-106, 1989.

[125] J. Choi, J. Sellen, and C.-K. Yap. Precision- sensitive Euclidean shortest path in 3-space. In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 350-359, 1995.

[126] J. Choi, J. Sellen, and C. K. Yap. Approximate Euclidean shortest paths in 3space. Internat. J. Comput. Geom. Appl., 7:271-295, August 1997.

[127] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Cambridge, MA, 2005.

[128] V. Chvatal. A combinatorial theorem in plane geometry. J. Combin. Theory Ser. B, 18:39-41, 1975.

[129] V. Chvatal. Linear Programming. W. H. Freeman, New York, 1983.

[130] K. L. Clarkson. Linear programming in O(n3d2) time. Inform. Process. Lett., 22:21-24, 1986.

[131] K. L. Clarkson. New applications of random sampling in computational geometry. Discrete Comput. Geom., 2:195-222,1987.

[132] K. L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. J. ACM, 42:488-499, 1995.

[133] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. Discrete Comput. Geom., 4:387-421, 1989.

[134] K. L. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. Discrete Comput. Geom., 4:423-432, 1989.

[135] R. Cole. Searching and storing similar lists. J. Algorithms, 7:202-220, 1986.

[136] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Proc. 2nd GI Conference on Automata Theory and Formal Languages. Lecture Notes in Computer Science, vol. 33, pages 134-183. Springer-Verlag, 1975.

[137] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. The MIT Press, Cambridge, MA, 2001.

[138] F. d'Amore and P. G. Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. Inform. Process. Lett., 44:255-259, 1992.

[139] G. B. Dantzig. Linear Programming and Extensions. Princeton University Press, Princeton, NJ, 1963.

[140] M. N. Demers. Fundamentals of Geographical Information Systems, 4th edn. Wiley, 2008.

[141] O. Devillers. Randomization yields simple $O(n\log^{*}n)$ algorithms for difficult $\Omega(n)$ problems. Internat. J. Comput. Geom. Appl., 2:97-111, 1992.

[142] O. Devillers and P. A. Ramos. Computing roundness is easy if the set is almost round. Internat. J. Comput. Geom. Appl., 12:229-248, 2002.

[143] T. Dey. Improved bounds for k-Sets and k-th levels. In Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci., pages 156-161, 1997.

[144] T. K. Dey. Improved bounds on planar k-sets and related problems. Discrete Comput. Geom., 19:373-383, 1998.

[145] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. Comput. Aided Geom. Design, 9:457-470, 1992.

[146] M. T. Dickerson, S. A. McElfresh, and M. H. Montague. New algorithms and empirical findings on minimum weight triangulation heuristics. In Proc. 11th Annu, ACM Sympos. Comput. Geom., pages 238-247, 1995.

[147] M. T. Dickerson and M. H. Montague. A (usually?) connected subgraph of the minimum weight triangulation. In Proc. 12th Annu. ACM Sympos. Comput. Geom., pages 204-213, 1996.

[148] G. L. Dirichlet. Uber die reduktion der positiven quadratischen formen mit drei unbestimmen ganzen zahlen. J. Reine Angew. Math., 40:209-227, 1850.

[149] D. Dobkin and D. Eppstein. Computing the discrepancy. In Proc. 9th Annu. ACM Sympos. Comput. Geom., pages 47-52, 1993.

[150] D. Dobkin and D. Mitchell. Random-edge discrepancy of supersampling patterns. In Graphics Interface'93, 1993.

[151] C. Duncan and M. Goodrich. Approximate geometric query structures. In D. Metha and S. Sahni, editors, Handbook of Data Structures and Applications, chapter 26. Chapman & Hall/CRC, 2005.

[152] D. Dutta, R. Janardan, and M. Smid. Geometric and Algorithmic Aspects of Computer-Aided Design and Manufacturing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 67. American Mathematical Society, 2005.

[153] M. E. Dyer. On a multidimensional search technique and its application to the Euclidean one-centre problem. SIAM J. Comput., 15:725-73 8, 1986.

[154] N. Dyn, D. Levin, and S. Rippa. Data dependent triangulations for piecewise linear interpolation. IMA Journal of Numerical Analysis, 10: 137-154, 1990.

[155] H. Ebarra, N. Fukuyama, H. Nakano, and Y. Nakanishi. Roundness algorithms using Voronoi diagrams. In Proc. First Canadian Conf. on Computational Geometry, page 41, 1989.

[156] W. E Eddy. A new convex hull algorithm for planar sets. ACM Trans. Math. Softw., 3:398-403 and 411-412, 1977.

[157] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.

[158] H. Edelsbrunner. Algorithms in Combinatorial Geometry. EATCS Monographs on Theoretical Computer Science, vol. 10. Springer-Verlag, 1987.

[159] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. J. Comput. Syst. Sci., 38:165-194, 1989. Corrigendum in 42 (1991), 249-251.

[160] H. Edelsbrunner, L. J. Guibas, J. Hershberger, R. Seidel, M. Sharir, J. Snoeyink, and E. Welzl. Implicitly representing arrangements of lines or segments. Discrete Comput. Geom., 4:433-466, 1989.

[161] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. SIAM J. Comput., 15:317-340, 1986.

[162] H. Edelsbrunner, G. Hating, and D. Hilbert. Rectangular point location in d dimensions with applications. Comput. J., 29:76-82, 1986.

[163] H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. Inform. Process. Lett., 13:177-181, 1981.

[164] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. ACM Trans. Graph., 9:66-104, 1990.

[165] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. SIAM J. Comput., 15:341-363, 1986.

[166] H. Edelsbrunner and M. H. Overmars. Batched dynamic solutions to decomposable searching problems. J. Algorithms, 6:515-542, 1985.

[167] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. Discrete Comput. Geom., 1:25-44, 1986.

[168] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. SIAM J. Comput., 22:418-429, 1993.

[169] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $O(n^{0.691})$ query time. Inform. Process. Lett., 23:289-293, 1986.

[170] H. E1Gindy and G. T. Toussaint. On triangulating palm polygons in linear time. In N. Magnenat-Thalmann and D. Thalmann, editors, New Trends in Computer Graphics, pages 308-317. Springer-Verlag, 1988.

[171] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In Proc. 8th Annu. ACM Sympos. Comput. Geom., pages 74-82, 1992.

[172] I. Emiris and J. Canny. A general approach to removing degeneracies. SIAM J. Comput., 24:650-664, 1995.

[173] D. Eppstein, M. Goodrich, and J. Sun. The skip quadtree: A simple dynamic data structure for multidimensional data. In Proc. 21st ACM Sympos. Comput. Geom., pages 296-205, 2005.

[174] P. Erdos, L. Lovasz, A. Simmons, and E. Straus. Dissection graphs of planar point sets. In J. N. Srivastava, editor, A Survey of Combinatorial Theory, pages 139-154. North-Holland, Amsterdam, Netherlands, 1973.

[175] I. D. Faux and M. J. Pratt. Computational Geometry for Design and Manufacture. Ellis Horwood, Chichester, U.K., 1979.

[176] U. Finke and K. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In Proc.11lth Annu. ACM Sympos. Comput. Geom., pages 119-126, 1995.

[177] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. Acta Inform., 4:1-9, 1974.

[178] S. Fisk. A short proof of Chvatal's watchman theorem. J. Combin. Theory Ser B, 24:374, 1978.

[179] J. D. Foley, A. van Dam, S. K, Feiner, and J. F. Hughes. Computer Graphics: Principles and Practice, 2nd edn. Addison-Wesley, Reading, MA, 1995.

[180] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In Proc. 8th Annu. ACM Sympos. Comput. Geom., pages 83-92,1992.

[181] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In Proc. 7th Annu. A CM Sympos. Comput. Geom., pages 334-341, 1991.

[182] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In Proc. 9th Annu. ACM Sympos. Comput. Geom., pages 163-172, 1993.

[183] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. Algorithmica, 2:153-174, 1987.

[184] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. ACM Trans. Graph., 3:153-174, 1984.

[185] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. Comput. Graph., 14:124-133, 1980. Proc. SIGGRAPH'80.

[186] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. Systematic Zoology, 18:259-278, 1969.

[187] J. Garcia-Lopez, P. A. Ramos, and J. Snoeyink. Fitting a set of points by a circle. Discrete Comput. Geom., 20:389-402, 1998.

[188] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. Inform. Process. Lett., 7:175-179, 1978.

[189] B. Gartner. A subexponential algorithm for abstract optimization problems. SIAM J. Comput., 24:1018-1035, 1995.

[190] S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. SIAM J. Comput., 20:888-910, 1991.

[191] J. E. Goodman and J. O'Rourke. Handbook of Discrete and Computational Geometry, 2nd edn. Chapman & Hall/CRC Press, 2004.

[192] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. Inform. Process. Lett., 1:132-133, 1972.

[193] R J. Green and B. W. Silverman. Constructing the convex hull of a set of points in the plane. Comput. J., 22:262-266, 1979.

[194] B. Grunbaum. Convex Polytopes. Wiley, New York, 1967.

[195] L. J. Guibas, J. Hershberger. D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. Algorithmica, 2:209-233, 1987.

[196] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica, 7:381-413, 1992.

[197] L. J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci., pages 100Ill, 1983.

[198] L. J. Guibas. D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In Proc. 5th Annu. ACM Sympos. Comput. Geom., pages 208-217, 1989.

[199] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci., pages 8-21, 1978.

[200] L. J. Guibas and R. Seidel. Computing convolutions by reciprocal search. Discrete Comput. Geom., 2:175-193, 1987.

[201] L. J. Guibas, M. Sharir, and S. Sifrony. On the general motion planning problem with two degrees of freedom. Discrete Comput. Geom., 4:491-521, 1989.

[202] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Trans. Graph., 4:74-123, 1985.

[203] L. J. Guibas and J. Stolfi. Ruler, compass and computer: the design and analysis of geometric algorithms. In R. A. Earnshaw, editor, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F, vol. 40, pages 111-165. Springer-Verlag, 1988.

[204] A. Guttman. R-trees: A dynamic index structure for spatial searching. In SIGMOD Conf., pages 47-57, 1984.

[205] D. Halperin. Algorithmic Motion Planning via Arrangements of Curves and of Surfaces. Ph.D. thesis, Computer Science Department, Tel-Aviv University, Tel Aviv, 1992.

[206] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 24. Chapman & Hall/CRC, 2004.

[207] D. Halperin and M. Sharir. Almost tight upper bounds for the single cell and zone problems in three dimensions. Discrete Comput. Geom., 14:385-410, 1995.

[208] D. Halperin and M. Sharir. Arrangements and their applications in robotics: Recent developments. In K. Goldbergs, D. Halperin, J.-C. Latombe, and R. Wilson, editors, Proc. Workshop on Algorithmic Foundations of Robotics. A. K. Peters, Boston, MA, 1995.

[209] D. Haussler and E. We1A. Epsilon-nets and simplex range queries. Discrete Comput. Geom., 2:127-151, 1987.

[210] J. Hershberger and S. Suri. Efficient computation of Euclidean shortest paths in the plane. In Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci., pages 508-517,1993.

[211] J. Hershberger and S. Suri. Off-line maintenance of planar configurations. J. Algorithms, 21:453-475, 1996.

[212] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. SIAM. J. Comput., 28:2215-2256, 1999.

[213] S. Hertel, M. Mantyla, K. Mehlhorn, and J. Nievergelt. Space sweep solves intersection of convex polyhedra. Acta Inform., 21:501-519, 1984.

[214] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In Proc. 4th Internat. Conj.' Found. Comput. Theory. Lecture Notes in Computer Science, vol. 158, pages 207-218. Springer-Verlag, 1983.

[215] K. Ho-Le. Finite element mesh generation methods: A review and classification. Comput. Aided Design, 20:27-38, 1988.

[216] C. Hoffmann. Geometric and Solid Modeling. Morgan Kauffmann, San Mateo, CA, 1989.

[217] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. Planning, Geometry, and Complexity of Robot Motion. Ablex, Publishing, Norwood, NJ, 1987.

[218] C. Icking, G. Rote, E. Welzl, and C. Yap. Shortest paths for line segments. Algorithmica, 10:182-200, 1993.

[219] H. Inagaki and K. Sugihara. Numerically robust algorithm for constructing constrained Delaunay triangulation. In Proc. 6th Canad. Conf. Comput. Geom., pages 171-176, 1994.

[220] R. Janardan and T. C. Woo. Design and manufacturing. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 55. Chapman & Hall/CRC, 2004.

[221] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. Inform. Process. Lett., 2:18-21, 1973.

[222] G. Kalai. A subexponential randomized simplex algorithm. In Proc. 24th Annu. ACM Sympos. Theory Comput., pages 475-482, 1992.

[223] M. Kallay. The complexity of incremental convex hull algorithms in $\mathbb{R}^d$. Inform. Process. Lett., 19:197, 1984.

[224] R. G. Karlsson. Algorithms in a restricted universe. Report CS-84-50, Univ. Waterloo, Waterloo, ON, 1984.

[225] R. G. Karlsson and J. I. Munro. Proximity on a grid. In Proceedings 2nd Symp. on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, vol. 182, pages 187-196. Springer-Verlag, 1985.

[226] R. G. Karlsson and M. H. Overmars. Scanline algorithms on a grid. BIT, 28:227-241, 1988.

[227] N. Karmarkar. A new polynomial-time algorithm for linear programming. Combinatorica, 4:373-395, 1984.

[228] M. Katz. 3-d vertical ray shooting and 2-d point enclosure, range searching, and arc shooting amidst convex fat objects. Comput. Geom. Theory Appl., 8:299-316, 1997.

[229] M. Katz, M. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. Comput. Geom. Theory Appl., 2:223-234, 1992.

[230] L. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. IEEE Trans. on Robotics and Automation, 12:566-580, 1996.

[231] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. Discrete Comput. Geom., 1:59-71, 1986.

[232] K. Kedem and M. Sharir. An efficient algorithm for planning collision-free translational motion of a convex polygonal object in 2-dimensional space amidst polygonal obstacles. In Proc. 1st Annu. ACM Sympos. Comput. Geom., pages 75-80,1985.

[233] K. Kedem and M. Sharir. An efficient motion planning algorithm for a convex rigid polygonal object in 2-dimensional polygonal space. Discrete Comput. Geom., 5:43-75, 1990.

[234] L. G. Khachiyan. Polynomial algorithm in linear programming. U.S.S.R. Comput. Math. and Math. Phys., 20:53-72, 1980.

[235] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. Internat. J. Robot. Res., 5:90-98, 1985.

[236] D. G. Kirkpatrick. Optimal search in planar subdivisions. SIAM J. Comput., 12:28-35, 1983.

[237] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan. Polygon triangulation in $O(n\log\log n)$ time with simple data structures. Discrete Comput. Geom., 7:329-346, 1992.

[238] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? SIAM J. Comput., 15:287-299, 1986.

[239] V. Klee. On the complexity of d-dimensional Voronoi diagrams. Archiv der Mathematik, 34:75-80, 1980.

[240] R. Klein. Abstract Voronoi diagrams and their applications. In Computational Geometry and its Applications. Lecture Notes in Computer Science, vol. 333, pages 148-157. Springer-Verlag, 1988.

[241] R. Klein. Concrete and Abstract Voronoi Diagrams. Lecture Notes in Computer Science, vol. 400. Springer-Verlag, 1989.

[242] R. Klein, K. Mehlhorn, and S. Meiser. Randomized incremental construction of abstract Voronoi diagrams. Comput. Geom. Theory Appl., 3:157-184, 1993.

[243] J.-C. Latombe. Robot Motion Planning. Kluwer Academic Publishers, Boston, 1991.

[244] C. L. Lawson. Transforming triangulations. Discrete Math., 3:365-372, 1972.

[245] C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, Math. Software III, pages 161-194. Academic Press, New York, 1977.

[246] D. Lee and A. Lin. Computational complexity of art gallery problems. IEEE Trans. Inform. Theory, 32:276-282, 1986.

[247] D. T. Lee. Proximity and reachability in the plane. Report R-831, Dept. Elect. Engrg., Univ. Illinois, Urbana, IL, 1978.

[248] D. T. Lee. Two-dimensional Voronoi diagrams in the L_p-metric. J. ACM, 27:604-618, 1980.

[249] D. T. Lee. On k-nearest neighbor Voronoi diagrams in the plane. IEEE Trans. Comput., C-31:478-487, 1982.

[250] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. SIAM J. Comput., 6:594-606, 1977.

[251] D. T. Lee and C. K. Wong. Quintary trees: a file structure for multidimensional database systems. ACM Trans. Database Syst., 5:339-353, 1980.

[252] D. T. Lee and C. K. Wong. Voronoi diagrams in $L_1(L_\infty)$ metrics with 2-dimensional storage applications. SIAM J. Comput., 9:200-211, 1980.

[253] D. T. Lee, C. D. Yang, and C. K. Wong. Rectilinear paths among rectilinear obstacles. Discrete Appl. Math., 70:185-215, 1996.

[254] J. van Leeuwen and D. Wood. Dynamization of decomposable searching problems. Inform. Process. Lett., 10:51-56, 1980.

[255] D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams. Discrete Comput. Geom., 19-31, 1987.

[256] C. Li, S. Pion, and C. K. Yap. Recent progress in exact geometric computation. J. Log. Algebr. Program., 64:85-111, 2005.

[257] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. Geographic Information Systems and Science, 2nd edn. Wiley, 2005.

[258] T. Lozano-P6rez. Automatic planning of manipulator transfer movements. IEEE Trans. Syst. Man Cybern., SMC-11: 681-698, 1981.

[259] T. Lozano-Perez. Spatial planning: A configuration space approach. IEEE Trans. Comput., C-32:108-120, 1983.

[260] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. Commun. ACM, 22:560-570, 1979.

[261] G. S. Lueker. A data structure for orthogonal range queries. In Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci., pages 28-34, 1978.

[262] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F, vol. 40, pages 307-325. Springer-Verlag, 1988.

[263] J. Matousek. Efficient partition trees. Discrete Comput. Geom., 8:315-334, 1992.

[264] J. Matousek. Reporting points in halfspaces. Comput. Geom. Theory Appl., 2:169-186, 1992.

[265] J. Matousek. Range searching with efficient hierarchical cuttings. Discrete Comput. Geom., 10:157-182, 1993.

[266] J. Matousek and O. Schwarzkopf. On ray shooting in convex polytopes. Discrete Comput. Geom., 10:215-232, 1993.

[267] J. Matousek, M. Sharir, and E. We1A. A subexponential bound for linear programming. Algorithmica, 16:498-516, 1996.

[268] J. Matousek, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. SIAM J. Comput., 23:154-169, 1994.

[269] H. A. Maurer and T. A. Ottmann. Dynamic solutions of decomposable searching problems. In U. Pape, editor, Discrete Structures and Algorithms, pages 17-24. Carl Hanser Verlag, Munchen, Germany, 1979.

[270] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL-80-9, Xerox Palo Alto Res. Center, Palo Alto, CA, 1980.

[271] E. M. McCreight. Priority search trees. SIAM J. Comput., 14:257-276, 1985.

[272] R. Mead. A relation between the individual plant-spacing and yield. Ann. of Bot. N.S., 30:301-309, 1966.

[273] N. Megiddo. Linear programming in linear time when the dimension is fixed. J. ACM, 31:114-127, 1984.

[274] K. Mehlhorn, S. Meiser, and C. O'Dunlaing. On the construction of abstract Voronoi diagrams. Discrete Comput. Geom., 6:211-224, 1991.

[275] K. Mehlhorn and S. N5her. Dynamic fractional cascading. Algorithmica, 5:215-241, 1990.

[276] K. Mehlhorn and M. H. Overmars. Optimal dynamization of decomposable searching problems. Inform. Process. Lett., 12:93-98, 1981.

[277] G. Meisters. Polygons have ears. Amer. Math. Monthly, 82:648-651, 1975.

[278] E. A. Melissaratos and D. L. Souvaine. Coping with inconsistencies: a new approach to produce quality triangulations of polygonal domains with holes. In Proc. 8th Annu. ACM Sympos. Comput. Geom., pages 202-211, 1992.

[279] V. Milenkovic. Robust construction of the Voronoi diagram of a polyhedron. In Proc. 5th Canad. Conf. Comput. Geom., pages 473-478, Waterloo, Canada, 1993.

[280] N. Miller and M. Sharir. Efficient randomized algorithms for constructing the union of fat triangles and pseudodiscs. Unpublished manuscript.

[281] J. S. B. Mitchell. Shortest paths among obstacles in the plane. In Proc. 9th Annu. ACM Sympos. Comput. Geom., pages 308-317, 1993.

[282] J. S. B. Mitchell. Shortest paths among obstacles in the plane. Internat. J. Comput. Geom. Appl., 6:309-332, 1996.

[283] J. S B. Mitchell and C. H. Papadimitriou. The weighted region problem: finding shortest paths through a weighted planar subdivision. J. ACM, 38:18-73, 1991.

[284] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. Algorithmica, 8:431-459, 1992.

[285] M. E. Mortenson. Geometric Modeling, 3rd edn. Industrial Press, New York, 2006.

[286] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. Theoret. Comput. Sci., 7:217-236, 1978.

[287] H. Muller. Rasterized point location. In Proceedings Workshop on Graph-theoretic Concepts in Computer Science, pages 281-293. Trauner Verlag, Linz, Austria, 1985.

[288] K. Mulmuley. A fast planar partition algorithm, I. In Proc. 29th Annu. IEEE Sympos. Found. Comput. Sci., pages 580-589, 1988.

[289] K. Mulmuley. A fast planar partition algorithm, I. Journal of Symbolic Computation, 10:253-280, 1990.

[290] K. Mulmuley. Computational Geometry: An Introduction Through Randomized Algorithms. Prentice Hall, Englewood Cliffs, NJ, 1994.

[291] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In Proc. 22nd Annu. ACM Sympos. Comput. Geom., pages 1-10, 2006.

[292] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. Comput. Graph., 24:115-124, August 1990. Proc. SIGGRAPH'90.

[293] J. Nievergelt and E P. Preparata. Plane-sweep algorithms for intersecting geometric figures. Commun. ACM, 25:739-747, 1982.

[294] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, Algorithmic Foundations of Geographic Information Systems. Lecture Notes in Computer Science, vol. 1340. Springer-Verlag, 1997.

[295] N. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In Proc. IJCAI, pages 509-520, 1969.

[296] C. O'Dunlaing and C. K. Yap. A "retraction" method for planning the motion of a disk. J. Algorithms, 6:104-111, 1985.

[297] A. Okabe, B. Boots, and K. Sugihara. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. John Wiley & Sons, Chichester, U.K., 1992.

[298] J. O'Rourke. Art Gallery Theorems and Algorithms. Oxford University Press, New York, 1987.

[299] M. H. Overmars. The Design of Dynamic Data Structures. Lecture Notes in Computer Science, vol. 156. Springer-Verlag, 1983.

[300] M. H. Overmars. Efficient data structures for range searching on a grid. J. Algorithms, 9:254-275, 1988.

[301] M. H. Overmars. Geometric data structures for computer graphics: an overview. In R. A. Earnshaw, editor, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F, vol. 40, pages 21-49. Springer-Verlag, 1988.

[302] M. H. Overmars. Point location in fat subdivisions. Inform. Process. Lett., 44:261-265, 1992.

[303] M. H. Overmars and J. van Leeuwen. Further comments on Bykat's convex hull algorithm. Inform. Process. Lett., 10:209-212, 1980.

[304] M. H. Overmars and J. van Leeuwen. Dynamization of decomposable searching problems yielding good worst-case bounds. In Proc. 5th GI Con Theoret. Comput. Sci. Lecture Notes in Computer Science, vol. 104, pages 224-233. Springer-Verlag, 1981.

[305] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. J. Comput. Syst. Sci., 23:166-204, 1981.

[306] M. H. Overmars and J. van Leeuwen. Some principles for dynamizing decomposable searching problems. Inform. Process. Lett., 12:49-54, 1981.

[307] M. H. Overmars and J. van Leeuwen.., Two general methods for dynamizing decomposable searching problems. Computing, 26:155-166, 1981.

[308] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. Inform. Process. Lett., 12:168-173,1981.

[309] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. In J. van Leeuwen, editor, Algorithms - ESA'94. Lecture Notes in Computer Science, vol. 855, pages 240-253. Springer-Verlag, 1994.

[310] M. H. Overmars and P. Svestka. A probabilistic learning approach to motion planning. In Algorithmic Foundations of Robotics, pages 19-38. A. K. Peters, Boston, MA, 1995.

[311] M. H. Overmars and C-K. Yap. New upper bounds in Klee's measure problem. SIAM J. Comput., 20:1034-1045, 1991.

[312] J. Pach and M. Sharir. On vertical visibility in arrangements of segments and the queue size in the Bentley-Ottman line sweeping algorithm. SIAM J. Comput., 20:460-470, 1991.

[313] J. Pach, W. Steiger, and E. Szerner6di. An upper bound on the number of planar k-sets. Discrete Comput. Geom., 7:109-123, 1992.

[314] J. Pach and G. Tardos. On the boundary complexity of the union of fat triangles. SIAM J. Comput., 31:1745-1760, 2002.

[315] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. CVGIP: Graph. Models Image Process., 56:304-311, 1994.

[316] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. Inform. Process. Lett., 20:259-263, 1985.

[317] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. Discrete Comput. Geom., 5:485-503, 1990.

[318] M. S. Paterson and F. E. Yao. Optimal binary space partitions for orthogonal objects. J. Algorithms, 13:99-113, 1992.

[319] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations. Discrete Comput. Geom., 16:419-453, 1996.

[320] M. Pocchiola and G. Vegter. The visibility complex. Internat. J. Comput. Geom. Appl., 6:279-308, 1996.

[321] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. Commun. ACM, 22:402-405, 1979.

[322] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. Commun. ACM, 20:87-93, 1977.

[323] F. P. Preparata and M. I. Shamos. Computational Geometry: An Introduction. Springer-Verlag, 1985.

[324] F. P. Preparata and R. Tamassia. Efficient point location in a convex spatial cell-complex. SIAM J. Comput., 21:267-280, 1992.

[325] E. Quak and L. Schumaker. Cubic spline fitting using data dependent triangulations. Comput. Aided Geom. Design, 7:293-302, 1990.

[326] E. A. Ramos. On range reporting, ray shooting and k-level construction. In Proc. 15th Annu. ACM Sympos. on Comput. Geom., pages 390-399, 1999.

[327] J. H. Reif and J. A. Storer. A single-exponential upper bound for finding shortest paths in three dimensions. J. ACM, 41:1013-1019, 1994.

[328] S. Rippa. Minimal roughness property of the Delaunay triangulation. Comput. Aided Geom. Design, 7:489-497, 1990.

[329] H. Rohnert. Shortest paths in the plane with convex polygonal obstacles. Inform. Process. Lett., 23:71-76, 1986.

[330] J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. Discrete Comput. Geom., 7:227-253, 1992.

[331] J.-R. Sack and J. Urrutia. Handbook of Computational Geometry. Elsevier Science Publishers, Amsterdam, 1997.

[332] H. Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In R. A. Earnshaw, editor, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F, vol. 40, pages 51-68. Springer-Verlag, 1988.

[333] H. Samet. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, Reading, MA, 1990.

[334] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA, 1990.

[335] H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Mateo, CA, 2006.

[336] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. Commun. ACM, 29:669-679,1986.

[337] J. B. Saxe and J. L. Bentley. Transforming static data structures to dynamic structures. In Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci., pages 148-168, 1979.

[338] H. W. Scholten and M. H. Overmars. General methods for adding range restrictions to decomposable searching problems. J. Symbolic Comput., T1-10, 1989.

[339] A. Schrijver. Theory of Linear and Integer Programming. Wiley-Interscience, New York, 1986.

[340] J. T. Schwartz and M. Sharir. On the "piano movers" problem 1: the case of a two dimensional rigid polygonal body moving amidst polygonal barriers. Commun. Pure Appl. Math., 36:345-398, 1983.

[341] J. T. Schwartz and M. Sharir. On the "piano movers" problem II: general techniques for computing topological properties of real algebraic manifolds. Adv. Appl. Math., 4:298-351, 1983.

[342] J. T. Schwartz and M. Sharir. A survey of motion planning and related geometric algorithms. In D. Kapur and J. Mundy, editors, Geometric Reasoning, pages 157-169. MIT Press, Cambridge, MA, 1989.

[343] J. T. Schwartz and M. Sharir. Algorithmic motion planning in robotics. In J. van Leeuwen, editor, Algorithms and Complexity. Handbook of Theoretical Computer Science, vol. A, pages 391-430. Elsevier, Amsterdam, 1990.

[344] R. Seidel. Output-size sensitive algorithms for constructive problems in computational geometry. Ph.D. thesis, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, 1986. Technical Report TR 86-784.

[345] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Comput. Geom. Theory Appl., 1:51-64, 1991.

[346] R. Seidel. Small-dimensional linear programming and convex hulls made easy. Discrete Comput. Geom., 6:423-434, 1991.

[347] R. Seidel. Convex hull computations. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 22. Chapman & Hall/CRC, 2004.

[348] J. Selig. Geometric Fundamentals of Robotics, 2nd edn. Monographs in Computer Science. Springer-Verlag, 2004.

[349] M. I. Shamos. Computational Geometry. Ph.D. thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1978.

[350] M. I. Shamos and D. Hoey. Closest-point problems. In Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci., pages 151-162, 1975.

[351] M. I. Shamos and D. Hoey. Geometric intersection problems. In Proc. 17th Annu. IEEE Sympos. Found. Comput. Sci., pages 208-215, 1976.

[352] M. Sharir. Algorithmic motion planning. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 47. Chapman & Hall/CRC, 2004.

[353] M. Sharir and P. K. Agarwal. Davenport-Schinzel Sequences and Their Geometric Applications. Cambridge University Press, New York, 1995.

[354] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In Proc. 9th Sympos. Theoret. Aspects Comput. Sci. Lecture Notes in Computer Science, vol. 577, pages 569-579. Springer-Verlag, 1992.

[355] T. C. Shermer. Recent results in art galleries. Proc. IEEE, 80:1384-1399, September 1992.

[356] J. Shewchuck. Delaunay Refinement Mesh Generation. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1997.

[357] J. Shewchuck. Delaunay refinement algorithms for triangular mesh generation. Comput. Geom. Theory Appl., 22:21-74, 2002.

[358] P. Shirley. Discrepancy as a quality measure for sample distributions. In E H. Post and W. Barth, editors, Proc. Eurographics'91, pages 183-194. Elsevier Science, Vienna, Austria, September 1991.

[359] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, and E Willemsen. Fundamentals of Computer Graphics, 2nd edn. A.K. Peters, 2005.

[360] R. Sibson. Locally equiangular triangulations. Comput. J., 21:243-245, 1978.

[361] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry, 2nd edn., chapter 34. Chapman & Ha1UCRC, 2004.

[362] A. van der Stappen. Motion Planning Amidst Fat Obstacles. Ph.D. thesis, Utrecht Univ., Utrecht, Netherlands, 1994.

[363] A. van der Stappen, M. Overmars, M. de Berg, and J. Vleugels. Motion planning in environments with low obstacle density. Discrete Comput. Geom., 20:561-587, 1998.

[364] A. F. van der Stappen, D. Halperin, and M. H. Overmars. The complexity of the free space for a robot moving amidst fat obstacles. Comput. Geom. Theory Appl., 3:353-373, 1993.

[365] A. E van der Stappen and M. H. Overmars. Motion planning amidst fat obstacles. In Proc. l0th Annu. ACM Sympos. Comput. Geom., pages 31-40, 1994.

[366] H. Sundar, D. Silver, N. Gagvani, and S. J. Dickinson. Skeleton based shape matching and retrieval. In Shape Modeling International, pages 130-142, 2003.

[367] S. Suri. Minimum link paths in polygons and related problems. Ph.D. thesis, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1987.

[368] R. E. Tarjan and C. J. Van Wyk. An O(nloglogn)-time algorithm for triangulating a simple polygon. SIAM J. Comput., 17:143-178, 1988. Erratum in 17:1061, 1988.

[369] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. Comput. Graph., 25:61-69, July 1991. Proc. SIGGRAPH '91.

[370] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. Comput. Graph., 21:153-162, 1987. Proc. SIGGRAPH'87.

[371] C. Toth. Binary space partition for line segments with a limited number of directions. SIAMJ. Comput., 32:307-325, 2003.

[372] C. Toth. A note on binary plane partitions. Discrete Comput. Geom., 30:3-16, 2003.

[373] C. Toth. Binary space partitions: Recent developments. In J. E. Goodman, J. Pach, and E. Welzl, editors, Combinatorial and Computational Geometry. MSRI Publications, vol. 52, pages 529-556. Cambridge University Press, 2005.

[374] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. Pattern Recogn., 12:261-268, 1980.

[375] V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection, layered segment trees and dynamization. J. Algorithms, 3:160-176, 1982.

[376] G. Vegter. The visibility diagram: A data structure for visibility problems and motion planning. In Proc. 2nd Scand. Workshop Algorithm Theory. Lecture Notes in Computer Science, vol. 447, pages 97-110. Springer-Verlag, 1990.

[377] R. C. Veltkamp. Shape matching: Similarity measures and algorithms. In Shape Modeling International, pages 188-197, 2001.

[378] R. Volpe and P. Khosla. Artificial potential with elliptical isopotential contours for obstacle avoidance. In Proc. 26th IEEE Conf. on Decision and Control, pages 180-185,1987.

[379] G. M. Voronoi. Nouvelles applications des parametres continus a la theorie des formes quadratiques. premier Memoire: Sur quelques proprietes des formes quadratiques positives parfaites. J. Reine Angew. Math., 133:97-178, 1907.

[380] G. M. Voronoi. Nouvelles applications des parametres continus a la theorie des formes quadratiques. deuxieme Memoire: Recherches sur les paralleloedres primitifs. J. Reine Angew. Math., 134:198-287, 1908.

[381] A. Watt. 3D Computer Graphics, 3rd edn. Addison-Wesley, Reading, MA, 1999.

[382] R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. Comput. Geom. Theory Appl., 36:66-87, 2007.

[383] E. We1zl. Constructing the visibility graph for n line segments in $O(n^2)$ time. Inform. Process. Lett., 20:167-171, 1985.

[384] E. Welzl. Partition trees for triangle counting and other range searching problems. In Proc. 4th Annu. ACM Sympos. Comput. Geom., pages 23-3 3, 1988.

[385] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, New Results and New Trends in Computer Science. Lecture Notes in Computer Science, vol. 555, pages 359-370. Springer-Verlag, 1991.

[386] D. E. Willard. Predicate-oriented database search algorithms. Ph.D. thesis, Aiken Comput. Lab., Harvard Univ., Cambridge, MA, 1978. Report TR-20-78.

[387] D. E. Willard. The super-b-tree algorithm. Report TR-03-79, Aiken Comput. Lab., Harvard Univ., Cambridge, MA, 1979.

[388] D. E. Willard. Polygon retrieval. SIAM J. Comput., 11:149-165, 1982.

[389] D. E. Willard. Log-logarithmic worst case range queries are possible in space O(n). Inform. Process. Lett., 17:81-89, 1983.

[390] D. E. Willard. New trie data structures which support very fast search operations. J. Comput. Syst. Sci., 28:379-394, 1984.

[391] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. J. ACM, 32:597-617, 1985.

[392] M. F. Worboys and M. Duckham. GIS, a Computing Perspective, 2nd edn. Chapman & Hall/CRC, 2004.

[393] A. C. Yao. A lower bound to finding convex hulls. J. ACM, 28:780-787, 1981.

[394] A. C. Yao, and F. F. Yao. A general approach to D-dimensional geometric queries. In Proc. 17th Annu. ACM Sympos. Theory Comput., pages 163-168, 1985.

[395] C. Yap. Towards exact geometric computation. Comput. Geom. Theory Appl., 7:3-23, 1997.

[396] C. Yap and E. Chang. Issues in the metrology of geometric tolerancing. In J. Laumond and M. Overmars, editors, Robotics Motion and Manipulation, pages 393-400, A. K. Peters, 1997.

[397] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. J. Comput. Syst. Sci., 402-18, 1990.

[398] D. Zhu and L-C Latombe. New heuristic algorithms for efficient hierarchical path planning. IEEE Trans. on Robotics and Automation, 7:9-20, 1991.

[399] G. M. Ziegler. Lectures on Polytopes. Graduate Texts in Mathematics, vol. 152. Springer-Verlag, 1994.

图表索引

观察结论、引理、定理及推论
索引

关键词索引

— B —

— C —

— D —

— F —

— G —

— H —

— I —

— K —

— L —

— M —

— N —

— P —

— Q —

— R —

— S —

— T —

— V —

— B —

— C —

— CH —

C

D

F

G

— H —

— J —

— K —

— L —

— M —

— N —

— O —

— P —

— Y —

— Z —

— ZH —

— SH —

— ZH —

— Z —